JPA Implementation Patterns: Bidirectional Associations vs. Lazy Loading

by Vincent Partington MVB · Aug. 25, 09 · Java Zone · Not set

This blog assumes that you are familiar with the Order/OrderLine example I introduced in the first two blogs of this series. If you are not, please review the example.

Consider the following code:

```
OrderLine orderLineToRemove = orderLineDao.findById(30);
orderLineToRemove.setOrder(null);
```

The intention of this code is to unassociate the OrderLine with the Order it was previously associated with. You might imagine doing this prior to removing the OrderLine object (although you can also use the @PreRemove annotation to have this done automatically) or when you want to attach the OrderLine to a different Order entity.

If you run this code you will find that the following entities will be loaded:

1. The OrderLine with id 30.
2. The Order associated with the OrderLine. This happens because the OrderLine.setOrder method invokes the Order.internalRemoveOrderLine method to remove the OrderLine from its parent Order object.
3. *All the other OrderLines that are associated with that Order!* The Order.orderLines set is loaded when the OrderLine object with id 30 is removed from it.

Depending on the JPA provider, this might take two to three queries. Hibernate uses three queries; one for each of the lines mentioned here. OpenJPA only needs two queries because

Depending on the JPA provider, this might take two to three queries. Hibernate uses three queries; one for each of the lines mentioned here. OpenJPA only needs two queries because it loads the Order with the OrderLine using an outer join.

However the interesting thing here is that all OrderLine entities are loaded. If there are a lot of OrderLines this can be a very expensive operation. Because this happens even when you are not interested in the actual contents of that collection, you are be paying a high price for keeping the bidirectional associations intact.

So far I have discovered three different solutions to this problem:

1. Don't make the association bidirectional; only keep the reference from the child object to the parent object. In this case that would mean removing the orderLines set in the Order object. To retrieve the OrderLines that go with an Order you would invoke the findOrderLinesByOrder method on the OrderLineDao. Since that would retrieve all the child objects and we got into this problem because there are a lot of those, you would need to write more specific queries to find the subset of child objects you need. A disadvantage of this approach is that it means an Order can't access its OrderLines without having to go through a service layer (a problem we will address in a later blog) or getting them passed in by a calling method.
2. Use the Hibernate specific @LazyCollection annotation to cause a collection to be loaded "extra lazily" like so:

   ```
   @OneToMany(mappedBy = "order", cascade = CascadeType.PERSIST, fetch = FetchType.LAZY)
   @org.hibernate.annotations.LazyCollection(org.hibernate.annotations.LazyCollectionOption.EXTRA)
   public Set<OrderLine> orderLines = new HashSet<OrderLine>();
   ```

   This feature should cause Hibernate to be able to handle very large collections. For example, when you request the size of the set, Hibernate won't load all the elements of the collections. Instead it will execute a SELECT COUNT(*) FROM ... query. But even more interestingly: modifications to the collection are queued instead of being directly applied. If any modifications are pending when the collection is accessed, the session is flushed before further work is done.

   While this works fine for the size() method, it doesn't work when you try and iterate over the elements of the set (see JIRA issue HHH-2087 which has been open for two and a half years). The extra lazy loading of the size also has at least two open bugs: HHH-1491 and HHH-3319. All this leads me to believe the extra lazy loading feature of Hibernate is a nice idea but not fully mature (yet?).

3. Inspired by the Hibernate mechanism of postponing operations on the collection until you really need them to be executed, I have modified the Order class to do something similar. First an operation queue has been added as a transient field to the Order class:

   ```
   private transient Queue<Runnable> queuedOperations = new LinkedList<Runnable>();
   ```

   Then the internalAddOrderLine and internalRemoveOrderLine methods have been changed so that they do not directly modify the orderLines set. Instead they create an instance of the appropriate subclass of the QueuedOperation class. That instance is initialized with the OrderLine object to add or remove and then placed on the queuesOperations queue:

```
public void internalAddOrderLine(final OrderLine line) {
        queuedOperations.offer(new Runnable() {
                public void run() { orderLines.add(line); }
        });
}

public void internalRemoveOrderLine(final OrderLine line) {
        queuedOperations.offer(new Runnable() {
                public void run() { orderLines.remove(line); }
        });
}
```

Finally the getOrderLines method is changed so that it executes any queued operations before returning the set:

```
public Set<? extends OrderLine> getOrderLines() {
        executeQueuedOperations();
        return Collections.unmodifiableSet(orderLines);
}

private void executeQueuedOperations() {
        for (;;) {
                Runnable op = queuedOperations.poll();
                if (op == null)
                        break;
                op.run();
        }
}
```

If there were more methods that need the set to be fully up to date, they would invoke the executeQueuedOperations method in a similar manner.

The downside here is that your domain objects get cluttered with even more "link management code" than we already had managing bidirectional associations. Abstracting out this logic to a separate class is left as an exercise for the reader. 😉

Of course this problem not only occurs when you have bidirectional associations. It surfaces any time you are manipulating large collections mapped with @OneToMany or @ManyToMany. Bidirectional associations just makes the cause less obvious because you think you are only manipulating a single entity.

You should not use the method described above at bullet #3 if you are cascading any operations to the mapped collection. If you postpone modifications to the collections your JPA provider won't know about the added or removed elements and will cascade operations to the wrong entities. This means that any entities added to a collection on which you have set

@CascadeType.PERSIST won't be persisted unless you explicitly invoke EntityManager.persist on them. On a similar note the Hibernate specific @org.hibernate.annotations.CascadeType.DELETE_ORPHAN annotation will only remove orphaned child entities when they are actually removed from Hibernate's PersistentCollection.

In any case, now you know what causes this performance hit and three possible ways to solve it. I am interested to hear whether you ran into this problem and how you solved it.

From http://blog.xebia.com
