JPA Implementation Patterns: Mapping Inheritance Hierarchies

by Vincent Partington MVB · **Sep. 15, 09 · Java Zone · Not set**

This week I will dwell on the choices offered when mapping inheritance hierarchies in JPA. JPA provides three ways to map Java inheritance hierarchies to database tables:

1. InheritanceType.SINGLE_TABLE - The whole inheritance hierarchy is mapped to one table. An object is stored in exactly one row in that table and the discriminator value stored in the discriminator column specifies the type of the object. Any fields not used in a superclass or a different branch of the hierarchy are set to NULL. This is the default inheritance mapping strategy used by JPA.
2. InheritanceType.TABLE_PER_CLASS - Every concrete entity class in the hierarchy is mapped to a separate table. An object is stored in exactly one row in the specific table for its type. That specific table contains column for all the fields of the concrete class, including any inherited fields. This means that siblings in an inheritance hierarchy will each have their own copy of the fields they inherit from their superclass. A UNION of the separate tables is performed when querying on the superclass.
3. InheritanceType.JOINED - Every class in the hierarchy is represented as a separate table, causing no field duplication to occur. An object is stored spread out over multiple tables; one row in each of the tables that make up its class inheritance hierarchy. The is-a relation between a subclass and its superclass is represented as a foreign key relation from the "subtable" to the "supertable" and the mapped tables are JOINed to load all the fields of an entity.

A nice comparison of the JPA inheritance mapping options with pictures, and including a description of the @MappedSuperclass option, can be found in the DataNucleus documentation.

Now the interesting question is: which method works best in what circumstances?

SINGLE_TABLE - Single table per class hierarchy

The SINGLE_TABLE strategy has the advantage of being simple. Loading entities requires querying only one table, with the discriminator column being used to determine the type of the entity. This simplicity also helps when manually inspecting or modifying the entities stored in the database.

A disadvantage of this strategy is that the single table becomes very large when there are a lot of classes in the hierarchy. Also, columns that are mapped to a subclass in the hierarchy should be nullable, which is especially annoying with large inheritance hierarchies. Finally, a change to any one class in the hierarchy requires the single table to be altered, making the SINGLE_TABLE strategy only suitable for small inheritance hierarchies.

TABLE_PER_CLASS - Table per concrete class

The TABLE_PER_CLASS strategy does not require columns to be made nullable, and results in a database schema that is relatively simple to understand. As a result it is also easy to inspect or modify manually.

A downside is that polymorphically loading entities requires a UNION of all the mapped tables, which may impact performance. Finally, the duplication of column corresponding to superclass fields causes the database design to not be normalized. This makes it hard to perform aggregate (SQL) queries on the duplicated columns. As such this strategy is best suited to wide, but not deep, inheritance hierarchies in which the superclass fields are not ones you want to query on.

JOINED - Table per class

The JOINED strategy gives you a nicely normalized database schema without any duplicate columns or unwanted nullable columns. As such it is best suited to large inheritance hierarchies, be the deep or wide.

This strategy does make the data harder to inspect or modify manually. Also, the JOIN operation needed to load entities can become a performance problem or a downright barrier to the size of your inheritance strategy. Also note that Hibernate does not correctly handle discriminator columns when using the JOINED strategy.

BTW, when using Hibernate proxies, be aware that lazily loading a class mapped with any of the three strategies above always returns a proxy that is an instanceof the superclass.

Are those all the options?

So to summarize you could say the following rules apply when choosing from JPA's standard inheritance mapping options:

- Small inheritance hierarchy -> SINGLE_TABLE.
- Wide inheritance hierarchy -> TABLE_PER_CLASS.
- Deep inheritance hierarchy -> JOINED.

But what if your inheritance hierarchy is *very* wide or *very* deep? And what if the classes in your system are modified often? As we found while building a persisted command framework and a flexible CMDB for our Java EE deployment automation product Deployit, the concrete classes at the bottom of a large inheritance hierarchy can change often. So these two questions often get a positive answer at the same time. Luckily there is one solution to both problems!

Using blobs

The first thing to note is that inheritance is a very large component of the object-relational impedance mismatch. And then question we should ask ourselves is: why are we even mapping all those often changing concrete classes to database tables? If object databases had really broken through, we might be better off storing those classes in such a database. As it is, relational database have inherited the earth so that is out of the question. It might also be that for a part of your object model the relational model actually makes sense because you want to perform queries and have the database manage the (foreign key) relations. But for some parts you are actually only interested in simple persistence of objects.

A nice example is the "persisted command framework" I mentioned above. The framework needs to store generic information about each command such as a reference to the "change plan" (a kind of execution context) it belongs to, start and end times, log output, etc. But it also needs to store a command object that represents the actual work to be done (an invocation of wsadmin or wlst or something similar in our case).

For the first part the hierarchical model is best suited. For the second part simple serialization will do. So we first define a simple interface that is implemented by the different command objects in our system:

```
public interface Command {
        void execute();
}
```

And then we create the entity that stores both the metadata (the data we want to store in a relational model) and the serialized command object:

```
@Entity
public class CommandMetaData {
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private int id;

        @ManyToOne
        private ChangePlan changePlan;

        private Date startOfExecution;

        private Date endOfExecution;

        @Lob
        private String log;

        @Lob
        @Column(name = "COMMAND", updatable = false)
```

```
        private byte[] serializedCommand;

        @Transient
        private Command command;

        public CommandMetaData(Command details) {
                serializedCommand = serializeCommand(details);
        }

        public Command getCommand() {
                if (command != null) {
                        command = deserializeCommand(serializedCommand);
                }
                return command;
        }

        [... rest omitted ...]
}
```

The serializedCommand field is a byte array that is stored as a blob in the database because of the @Lob annotation. The column name is explicitly set to "COMMAND" to prevent the default column name of "SERIALIZEDCOMMAND" from appearing in the database schema.

The command field is marked as @Transient to prevent it from being stored in the database.

When a CommandMetaData object is created, a Command object is passed in. The constructor serializes the command object and stores the results in the serializedCommand field. After that the command cannot be changed (there is no setCommand() method), so the serializedCommand can be marked as not updatable. This prevents that pretty big blob field from being written to the database every time another field of the CommandMetaData (such as the log field) is updated.

Every time the getCommand method is invoked, the command is deserialized if needed and then it is returned. The getCommand could be marked synchronized if this object were used in multiple concurrent threads.

Some things to note about this approach are:

- The serialization method used influences the flexibility of this approach. Standard Java serialization is simple but does not handle changing classes well. XML can be an alternative but that brings its own versioning problems. Picking the right serialization mechanism is left as an exercise for the reader.
- Although blobs have been around for a while, some databases still struggle with them. For example, using blobs with Hibernate and Oracle can be tricky.
- In the approach presented above, any changes made to the Command object after it has been serialized will not be stored. Clever use of the @PrePersist and @PreUpdate

lifecycle hooks could solve this problem.

This semi-object database/semi-relational database approach to persistence worked out quite well for us. I am interested to hear whether other people have tried the same approach and how they fared. Or did you think of another solution to these problems?

From http://blog.xebia.com

Like This Article? Read More From DZone

Proposed Jakarta EE Design Principles

Advanced Research Initiatives: Focus on the Future

How to Configure NGINX High Availability Cluster Using Pacemaker on Ubuntu 16.04

Free DZone Refcard

Getting Started With Kotlin

Topics:

Learn more about Kotlin
JetBrains

Building Real-Time Data Pipelines with a 3rd Generation Stream Processing Engine
Hazelcast

Learn how to refactor a monolithic application to work your way toward a scalable and resilient microsystem.
Lightbend
