Java Persistence API (JPA). Parte 1

Introducción al Mapeo Objeto Relacional

Resúmen

Este primer artículo, de una serie sobre el estándar JPA, introduce los conceptos básicos del Mapeo Objeto Relacional (ORM) necesarios para comprender el diseño y funcionamiento de la Java Persistence API.

El contenido presentado en esta introducción es simplemente un resúmen de los conceptos teóricos presentados en los excelentes whitepapers de Scott W. Ambler listados al final en la sección de referencias.

En un próximo artículo presentaremos con ejemplos el uso básico de la Java Persistence API en aplicaciones de escritorio.

¿Qué es ORM?

"Object-Relational Mapping" es una técnica de programación que resuelve el problema de "convertir" datos entre los sistemas de tipos incompatibles de un motor de base de datos relacional (RDBMS) y un lenguaje orientado a objetos, en nuestro caso Java. El objetivo es crear una capa de persistencia en la forma de una base de datos orientada a objetos (OODB) “virtual” que oculte al programador los detalles de persistencia de los datos en el RDBMS. Existen actualmente varias implementaciones Java, tanto gratuitas como comerciales, de las cuales se listan algunas al final del artículo.

Problema básico

La manipulación de datos en Programación Orientada a Objetos (OOP) involucra la manipulación de Objetos con propiedades simples y asociaciones de agregación, composición, herencia, etc. a otros objetos (datos). Pero además, y no menor, los objetos definen comportamiento mediante métodos que manipulan su estado interno además de relaciones de herencia y la propiedad de polimorfismo.

Por su parte los RDBMS manipulan información escalar (int, string, etc.) organizada en tablas y no definen comportamiento ni relaciones de herencia asociado a las entidades de datos (registros en las tablas de la base de datos). Es el programador quien debe convertir, si no cuenta con un framework adecuado, objetos en datos tabulares y viceversa, lo cual puede resultar en una tarea muy compleja.

Si bien la utilización de Bases de Datos Orientadas a Objetos (OODBMS) permitiría solucionar este problema, las mismas no han sido tan exitosas como los RDBMS. Esto ha llevado al diseño de los llamados frameworks de persistencia Objeto/Relacional basados en las técnicas conocidas como Mapeo Objeto/Relacional (ORM por la sigla en inglés de Object Relational Mapping).

Las técnicas de ORM permiten automatizar procesos que trasladan objetos a formas almacenables en tablas y viceversa, preservando los atributos de los objetos. Para esto se basan en la utilización de metadatos de "mapping" que especifican la información necesaria para que un framework de persistencia ORM pueda efectuar de forma automática la conversión de datos entre el sistema relacional y el sistema orientado a objetos.

"Object/Relational Impedance Mismatch"

Scott W. Ambler ha identificado y le ha dado este nombre al conjunto de dificultades conceptuales y técnicas encontradas frecuentemente cuando se utiliza un RDBMS por un programa escrito en un lenguaje OO. No entraremos en detalles, que se pueden encontrar muy bien explicados en los papers referido, sino que presentaremos brevemente las dificultades principales.

En OOP es muy utilizada la técnica de encapsulamiento de datos, la cual consiste en ocultar la representación interna del estado de los objetos.
En un RDBMS los campos de las tablas se exponen directamente, aunque es posible especificar ciertas "restricciones" sobre los valores posibles de los mismos.

Mapear atributos encapsulados directamente a campos de tablas relacionales redunda en bases de datos frágiles (ej. pérdida de validaciones al cargar el atributo desde la base de datos). Una solución desde el lado OO es mapear propiedades (pares de métodos setter / getter) a campos de la base de datos. Al cargar el objeto desde la base de datos no se establecerá directamente el valor del atributo en cuestión sino que se usará el método setter del mismo, permitiendo que se apliquen todas las validaciones y/o conversiones de datos programadas en el mismo. Es importante tener en cuenta que un par de métodos setter/getter no necesariamente tienen que corresponder a un atributo simple del objeto (esta es la gracia del ocultamiento de información) dado que no necesariamente la representación interna de la propiedad tiene que ser un atributo.

Algunos conceptos de Orientación a Objetos no soportados por los RDBMS incluyen:

- Interfaces
- Herencia y polimorfismo
- Asociaciones de distintos tipos (con diferente semántica aunque estructuralmente se representen de la misma forma): Composición, Agregación, etc.

En un RDBMS los propios datos datos son la interfaz mientras que en OOP el comportamiento de un objeto define su interfaz.

En un RDBMS los campos de una tabla son accedidos y alterados mediante operadores relacionales predefinidos, ejemplo: SELECT, INSERT, UPDATE, mientras que en OOP cada clase puede proveer sus propias interfaces y prácticas para la alteración de su estado interno.

En cuanto a las relaciones entre entidades, en un RDBMS no existe una asociación fuerte entre operaciones (acciones) y datos (entidades). No existe concepto de Clase. En OOP se

fomenta una fuerte asociación entre acciones (operaciones) y entidades sobre las que estas operan (Concepto de Clase).

En cuanto a la unicidad de instancias/filas, en un RDBMS la filas requieren generalmente claves primarias mientras que en OOP no se requiere un identificador único visible externamente.

En cuanto a la identidad de instacias/filas, en un RDBMS en general no puedo tener dos tuplas con idéntico estado (PK, claves únicas) mientras que en OOP dos objetos no inmutables se considera que tienen identidad única (Dos instancias con idéntico estado no son el mismo objeto).

En cuanto a la normalización, en un RDBMS la normalización se utiliza para mejorar almacenamiento y performance en el acceso a los datos, mientras que en OOP no se aplica normalización relacional. En general los datos OO están desnormalizados en términos relacionales.

En cuanto a la herencia, un RDBMS no soporta herencia de entidades mientras que en OOP es una práctica común que permite la reutilización de la definición de clases más generales en clases más específicas y el polimorfismo.

En cuanto a los objetivos de diseño de ambos modelos de información, un RDBMS hace foco en el comportamiento del sistema en ejecución (eficiencia, adaptabilidad, tolerancia a fallas, integridad lógica, etc.) mientras que en OOP se priorizan otras propiedades enfocadas en asegurar al desarrollador estructuras razonables de programas (mantenibilidad, comprensión, extensibilidad, reusabilidad, seguridad).

Mapeando objetos a RDBMS

Es muy común encontrar aplicaciones OO manipulando datos en bases de datos relacionales, mediante el uso de técnicas de mapeo por metadatos.

Dentro de las consideraciones a tener en cuenta encontramos:

- Un atributo o propiedad podría mapearse a cero o mas columnas en una tabla de un RDBMS.
- Algunos atributos o propiedades de los objetos no son persistentes (calculados por la aplicación)
- Algunos atributos de un objeto son también objetos (Cliente --> Dirección) y esto refleja una asociación entre dos clases que deben tener sus propios atributos mapeados.

En principio identificaremos dos tipos de metadatos de mapping (property mapping y relationship mapping).

Property mapping: Mapping que describe la forma de persistir una propiedad de un objeto.

Relationship mapping: Mapping que describe la forma de persistir una relación (asociación, agregación, o composición) entre dos o mas objetos.

El mapping mas simple es un “property mapping” de un atributo simple a una columna de su mismo tipo. Pero, excepto para casos triviales, no siempre es tan sencillo como mapear una clase a una tabla en una relación uno a uno donde cada propiedad de la clase corresponde a un campo de una tabla en la base de datos.

Por otra parte para soportar la conversión entre los dos modelos es necesario incorporar al modelo OO lo que se conoce como información "Shadow". Esto es, cualquier dato extra (agregado al modelo OO original y sin significado para el negocio) que necesiten mantener los objetos para poder persistirse, como por ejemplo identificación de las propiedades que corresponden a la clave primaria en la base de datos, marcas que permitan el control de intentos de modificación concurrente de los datos en la base (timestamps o números de versión de los datos), atributos especiales que indiquen si el objeto ya fue persistido (para determinar si se usa INSERT o UPDATE), etc.

Los metadatos que configuran el mapeo pueden llegar a ser muy complejos y es necesario comprender muy bien que se está configurando y las implicaciones de usar diferentes alternativas.

Mapeando relaciones de herencia:

Como ya comentamos, los RDBMS no soportan herencia en forma nativa y por lo tanto es necesario mapear.

La decisión fundamental está en la respuesta a la pregunta ¿cómo organizo los atributos heredados en el modelo de datos?

Para ilustrar la problemática, y las distintas estrategias, utilizaremos el ejemplo presentado por Ambler para mostrar de forma sencilla la evolución de un modelo de información Orientado a Objetos.

Consideremos el siguiente modelo orientado a objetos:

```
abstract class Persona {
    public String nombre;
}

class Cliente extends Persona {
    public String preferencias;
}

class Empleado extends Persona {
    public double salario;
}
```

y una posible evolución del mismo al incorporar la clase Ejecutivo como especialización de la clase Empleado en una etapa posterior:

```
class Ejecutivo  extends Empleado {
    public double bonificacion;
}
```

Entre las estrategias posibles para mapear las relaciones de herencia encontramos:

- Mapear la jerarquía de clases a una sola tabla
- Mapear cada clase concreta a su propia tabla
- Mapear cada clase a su propia tabla

Veremos las básicas de cada una de ellas y usaremos el ejemplo para presentar sus fortalezas y debilidades.

Estrategia 1: Mapear la jerarquía de clases a una sola tabla

Se mapean todos los atributos de todas las clases de la jerarquía a una sola tabla.

En el ejemplo:

Para el modelo original se crea una tabla conteniendo los campos de todas las clases:

```
tabla PERSONA [
    ID : Clave primaria.
    DISCRIMINADOR: VARCHAR
    NOMBRE: VARCHAR
    PREFERENCIAS: VARCHAR
    SALARIO: DECIMAL
]
```

Vemos que además es necesario agregar en la clase Persona un campo para la clave primaria (Información "Shadow") y un campo discriminador en la tabla que permita determinar de que clase es la instancia persistida.

El discriminador puede contener simplemente el nombre completo de la clase concreta del objeto que se ha grabado.

Por ejemplo, si nuestras clases están en el paquete test y grabo una instancia de Empleado, el campo discriminador contendrá el valor "test.Empleado", lo cual me permite reconstruir de forma eficiente el objeto desde la base de datos.

Ahora, que sucede con la evolución del modelo. Al agregar la clase Ejecutivo es necesario agregar el campo "BONIFICACION: DECIMAL" a la tabla. Y definirlo como nullable!! porque los registros correspondientes a Cliente o Empleado no definen un valor para la bonificación (no tiene sentido).

Ventajas

Es un enfoque simple que permite agregar nuevas clases de forma sencilla (solo agregar columnas).

Soporta polimorfismo mediante inspección del valor del campo discriminador.
Provee acceso muy eficiente a los datos (Sin JOINS).
La creación de reportes ad-hoc es muy sencilla ya que todos los datos están en una sola tabla en una tabla.

Desventajas
Alto acoplamiento con la jerarquía de clases ya que todas se persisten en la misma tabla. Un cambio en la jerarquía afecta a todas.
Potencial desperdicio de espacio en la BD, por campos irrelevantes a la clase. Por ejemplo al tener que grabar un valor por defecto para la bonificacion al grabar un Cliente.
Impide la definición de restricciones dado que hay campos que deben quedar en null o en un valor por defecto si no corresponden a la clase de la instancia que se está grabando.
Si la jerarquía es muy grande obtengo tablas muy grandes (con muchos campos).

¿Cuándo usar?
Buena estrategia para clases simples o jerarquías poco profundas, dónde los tipos de la jerarquía no se solapan. **Estrategia 2: Mapear cada clase concreta a su propia tabla**
Se crea una tabla por cada clase concreta, redundando todos los atributos y no se mapean clases abstractas.

En el ejemplo: dado que la clase Persona es abstracta no creamos una tabla para la misma.
Creamos las siguientes tablas:

```
tabla CLIENTE [
    ID : Clave primaria.
    NOMBRE: VARCHAR
    PREFERENCIAS: VARCHAR
]

tabla EMPLEADO [
    ID : Clave primaria.
    NOMBRE: VARCHAR
    SALARIO: DECIMAL
]
```

Nuevamente es necesario agregar un campo para la clave primaria en la clase Persona (información "shadow").

Al evolucionar el modelo se agrega una tabla nueva:

```
tabla EJECUTIVO [
    ID : Clave primaria.
    NOMBRE: VARCHAR
    SALARIO: DECIMAL
   BONIFICACION: DECIMAL
]
```

A simple vista podemos observar que un Empleado ascendido a ejecutivo requerirá copiar sus datos de la tabla EMPLEADO a la tabla EJECUTIVO, incluso es posible que sea necesario mantener la información redundante en las dos tablas para no afectar los procesos de negocios que manipulan Empleado.

Ventajas
El reporting ad-hoc sigue siendo sencillo ya que todos los datos necesarios sobre una clase particular se encuentran almacenados en una sola tabla.
Buena performance para acceder los datos de un objeto simple (sin asociaciones no es necesario el JOIN).

Desventajas
Modificar una clase implica modificar su tabla y las tablas de todas sus subclases (ej. agregar una columna direccion en la Persona implica modificar todas las tablas).
Cada vez que un objeto cambia su rol (se contrata un Cliente como Empleado) es necesario copiar todos sus datos a la tabla apropiada.
Dificultad para soportar múltiples roles y mantener integridad de datos.

¿Cuándo usar?
Cuando la jerarquía de clases es muy estable (las clases raramente cambian).
Cuando no existe solapamiento de tipos. **Estrategia 3: Mapear cada clase a su propia tabla**
Se crea una tabla por cada clase, sin redundar los atributos de las superclases.
Para recuperar una instancia es necesario recurrir a JOINS por clave primaria que en las tablas de las subclases son claves foraneas.

En el ejemplo:

```
tabla PERSONA [
    ID : Clave primaria.
    NOMBRE: VARCHAR
]

tabla  CLIENTE [
    ID : Clave primaria. // Foreign key a PERSONA.ID
    PREFERENCIAS: VARCHAR
]

tabla EMPLEADO [
    ID : Clave primaria. // Foreign key a PERSONA.ID
    SALARIO: DECIMAL
]
```

Nuevamente es necesario agregar un campo para la clave primaria en la clase Persona (información "shadow").

Al evolucionar el modelo se agrega una tabla nueva:

```
tabla EJECUTIVO [
    ID : Clave primaria. // Foreign key a EMPLEADO.ID
    BONIFICACION: DECIMAL
]
```

Para recuperar los datos de un Ejecutivo es necesario realizar el JOIN entre Persona, Empleado y Ejecutivo.

Ventajas
Fácil de entender (mapping uno a uno).
Soporta muy bien el polimorfismo agregando un campo DISCRIMINADOR en la tabla PERSONA, que permita identificar los JOINS a realizar sin escanear las tablas buscando los datos.
Fácil de modificar superclases y agregar nuevas subclases (implica modificar o agregar una sola tabla).
Modelo muy normalizado en el cual el tamaño de los datos crece en proporción directa al número de objetos (instancias) persistidos.

Desventajas
Muchas tablas en la BD (una por clase + tablas de relación).
Escritura y lectura de datos potencialmente menos eficiente (dividir las tablas en discos diferentes puede ayudar).
Reporting ad-hoc dificultoso debido a la necesidad de JOINS, a menos que se agreguen vistas para simular las tablas deseadas.

¿Cuándo usar?
Cuando hay solapamiento de tipos.
Cuando las modificaciones a las clases son frecuentes.
Cuando se quiere evitar la redundancia de datos. **Mapeando asociaciones**
Eisten tres tipos de asociaciones entre objetos

- Asociación
- Agregación
- Composición

Por ahora las trataremos igual, aunque existen diferencias en el manejo de las restricciones. Las cuales se pueden clasificar de acuerdo a dos categorías ortogonales:

En base a la multiplicidad:

- One-to-one
- One-to-many (many-to-one según desde donde se lea)

- Many-to-many

En base a la dirección

- Unidireccionales
- Bidireccional

En la base de datos las relaciones se mantienen mediante el uso de Foreign Keys

- One-to-one: FK implementada en una de las tablas
- One-to-many:FK desde la “one table” a la “many table”
- Many-to-many: es necesario incorporar una tabla asociativa (o de relación)

En cuanto a la direccionalidad, todas las relaciones en la base de datos relacional son efectivamente bidireccionales.

No entraremos en profundidad en los conceptos de mapeo de relaciones, se puede encontrar toda la información en los papers de Ambler, pero comentaremos brevemente algunas consideraciones.

Una configuración importante es si los objetos asociados a otro se leen automáticamente al leer el mismo. Cargar un objeto y todos sus objetos asociados automáticamente en memoria puede llegar a impactar negativamente en el rendimiento del sistema. Para esto se usan técnicas de "lazy loading" que consisten en cargar los objetos asociados a demanda a medida que son solicitados. Si una asociación nunca es solicitada entonces nunca se lee de la base de datos. Esta inteligencia es bastante compleja de implementar manualmente.

Otra consideración importante tiene que ver con el almacenamiento de colecciones secuenciales de objetos asociados a un objeto padre. En este caso es necesario almacenar en la base la información (por ejemplo un ordinal numérico) que permita recuperar estos objetos en la secuencia correcta.

Optimizaciones de rendimiento

A nivel de la base de datos puede ser necesario cambiar el esquema, usualmente desnormalizando porciones del mismo, cambiar los tipos de las columnas clave (numéricos son mas efectivos que los strings), reducir la cantidad de columnas que componen una clave, introducir índices o inlcuso introducir procedimientos almacenados que trasladen a la base ciertos procesos.

También puede ser necesario agregar cachés en memoria que minimizen la cantidad de accesos a la base de datos a los mínimos necesarios.

En cuanto a la optimización de los mapeos siempre hay que tener en cuenta que tengo:

- 4 maneras de mapear herencia (vimos 3)
- 2 maneras de mapear relaciones one to one (no las vimos)
- 4 maneras de mapear atributos de clase (no las vimos)

Siempre hay que tener en cuenta que cada vez que se cambia la estrategia de mapping puede ser necesario cambiar el esquema de objetos, el esquema de base de datos o ambos. Por lo que la recomendación es elegir muy bien las estrategias a utilizar.

Impacto sobre el modelo de objetos

En cuanto al impacto de utilizar una estrategia de persistencia ORM sobre el modelo de objetos se distinguen los siguientes puntos:

- Se debe agregar información "shadow" (ejemplo: campo para el identificador único)
- Puede ser necesario hacer refactorings sobre el modelo original para mejorar el rendimiento de la base de datos.
- Puedo encontrarme con bases de datos legadas no diseñadas para ORM.
- Es bueno encapsular en una capa de data access objects el mecanismo de acceso a base de datos.
- Es necesario implementar mecanismos de control de concurrencia en aplicaciones multitarea en las que puedan estar afectándose los mismos datos desde procesos distintos.

- Hay que tener en cuenta las estrategias para recuperar los objetos de la base de datos relacional.
- Implementar integridad referencial (en la aplicación y en la BD).
- Es recomendable utilizar herramientas especializadas de reporting.
- Generalmente es necesario implementar cachés de objetos para minimizar los accesos a la base de datos.

Requerimientos de la capa de persistencia

La primer parte del artículo se enfocó en presentar algunas de las complejidades de persistir modelos de objetos en bases de datos relacionales. En esta sección presentaremos los principales requerimientos para un buen framework de persistencia ORM.

Una capa de persistencia encapsula el comportamiento necesario para persistir objetos. O sea: leer, escribir y borrar objetos en el almacenamiento persistente (base de datos).

Un buen framework de persistencia debería proveer de forma relativamente sencilla:

- Soporte para diversos tipos de mecanismos de persistencia (archivos planos, RDBMS, OODBMS, etc.)

- Encapsulamiento total del mecanismo de persistencia.
- Acciones sobre múltiples objetos (asociaciones, búsquedas, etc.).
- Transacciones: planas o anidadas, locales o distribuidas
- Extensibilidad: permitir agregar nuevas clases (entidades) de forma sencilla.
- Soportar la manipulación automática de identificadores de objetos (OIDs).
- Cursores que permitan la lectura incremental de objetos desde la base de datos.
- Soporte para proxies que habiliten las técnicas de "lazy loading".
- Registros (Record Sets) para operaciones de bajo nivel.
- Soporte para múltiples arquitecturas (Desktop, Enterprise).
- Soporte para diferentes versiones de una base de datos o diferentes proveedores en forma transparente a la aplicación.
- Soporte para el manejo eficiente de ,últiples conexiones a diferentes BD (pooles de conexiones).
- Soporte para drivers de base de datos nativos y no nativos.
- Permitir ejecutar consultas SQL si es necesario
- Proveer un lenguaje para consultas Orientadas a Objetos, mas naturales al modelo de información de la aplicación, y trasalación automática de estas a SQL al momento de ejecutar la consulta.

Implementaciones

- Hibernate
- Castor
- Apache OJB
- Oracle TopLink Essentials
- Apache Open JPA

Referencias

- Whitepaper "The Design of a Robust Persistence Layer for Relational Databases", Scott W. Ambler, http://www.agiledata.org
- Whitepaper "Mapping Objects to Relational Databases", Scott W. Ambler, http://www.agiledata.org