Armazenando Dados em Aplicações Java

Parte 3 de 3: Analisando as opções

Hua Lin Chang Costa, hualin@cos.ufrj.br, COPPE/UFRJ.
Leonardo Gresta Paulino Murta, leomurta@ic.uff.br, IC/UFF.
Vanessa Braganholo, braganholo@dcc.ufrj.br, DCC/UFRJ.

Quais são os problemas e as opções disponíveis para armazenamento de dados em aplicações Java de verdade? Este artigo é o terceiro de uma trilogia, e complementa a resposta da parte "quais são as opções disponíveis" dessa pergunta com uma analise mais detalhada entre as principais opções.

Quando estamos em um ambiente acadêmico e precisamos fazer um programa para alguma disciplina ou trabalho final de curso, o armazenamento dos dados é de pouca importância, e soluções como serialização normalmente são suficientes. Contudo, depois de formados precisaremos lidar com sistemas reais, em ambientes reais, e a solução anterior deixa de ser satisfatória. Este artigo é o terceiro de uma trilogia, e exemplifica como algumas das opções para tratar o problema funcionam, viabilizando a escolha da opção mais adequada para o seu problema específico.

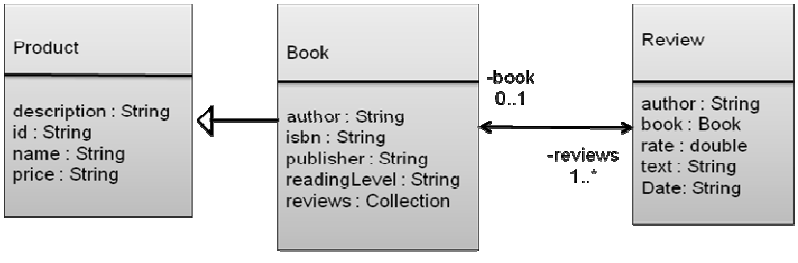
No primeiro artigo da trilogia analisamos um caso real e fizemos uma pesquisa para identificar quais seriam os principais requisitos de armazenamento de dados em aplicações Java. Naquele artigo, chegamos aos seguintes requisitos principais: (1) compatibilidade entre diferentes versões da aplicação, (2) otimização em termos de desempenho, (3) confiabilidade, (4) facilidade de manutenção e evolução e (5) não intrusão. No segundo artigo da trilogia vimos alguns dos principais *frameworks* disponíveis para tratar o problema de armazenamento de dados em Java: XStream, Castor, Torque, Hibernate, JPOX e TopLink Essentials. Também percebemos que alguns desses *frameworks* são aderentes às especificações JDO ou JPA.

Esse artigo seleciona as opções aderentes a especificação JDO ou JPA (atendendo ao requisito 4), ou seja, JPOX, Hibernate e TopLink Essentials, para fazer uma análise mais detalhada. Para isso, projetamos um modelo simples, mas que envolve diferentes aspectos importantes da orientação a objetos. Esse modelo é armazenado e recuperado de acordo com cada uma das especificações. A seguir, apresentamos o modelo criado para a análise em questão e em seguida apresentamos como a camada de persistência poderia ser implementada por cada uma das especificações.

Modelo exemplo utilizado

A análise das opções selecionadas foi realizada sobre um modelo de classes simplificado, mas que representa pontos importantes do mapeamento objeto-relacional. Assim, o modelo exemplo, mesmo que de forma simples, apresenta relacionamentos como associações e herança entre as classes para que esses pontos possam ser avaliados em cada um dos *frameworks*.

O modelo de classes adotado para aplicação do estudo apresenta um domínio onde é representado o artefato livro (Book), que é uma especialização de um artefato mais genérico, produto (Product). Com o objetivo de cobrir também relacionamentos do tipo associação, o modelo representa uma relação entre um livro e diferentes resenhas (Review). A figura ao lado ilustra o modelo de classes utilizado.



Modelo de classes utilizado para análise dos *frameworks*

Especificação JDO

O *framework* JPOX na versão 1.2 provê suporte para diferentes SGBDs e para ambas as especificações JDO e JPA. Para a implementação da camada de persistência sobre o modelo de classes exemplo, o JPOX foi utilizado como sendo uma implementação da especificação JDO 2.0. O fato de o JPOX ser a implementação de referência dessa especificação reforça essa escolha.

No momento da execução desse estudo, no final de 2007, a versão da especificação JDO 2.0 ainda não continha a documentação oficial do modelo de declaração dos metadados através do uso de anotações (ou seja, marcações diretamente em código, como discutido mais a frente). Por isso foram utilizados documentos XML para configuração do mapeamento-relacional.

No modelo exemplo existem duas classes que estão relacionadas: 'Book' e 'Review'. O mecanismo de mapeamento objeto-relacional provido pelo JDO permite duas formas de persistência para esse tipo de mapeamento: o uso de tabela associativa ou o uso de chave estrangeira. No uso de tabela associativa, uma tabela extra é usada para relacionar as duas tabelas, enquanto que no uso de chave estrangeira a relação entre as classes é representada através de um atributo. No modelo exemplo, um objeto Review só pode ser adicionado uma única vez em determinada coleção de um objeto Book, então será aplicada a solução que faz uso de chave estrangeira para evitar a criação de uma tabela extra.

Outro tipo de relacionamento existente no modelo de classes exemplo é a herança A especificação JDO define três formas possíveis de realizar esse mapeamento. Na primeira uma nova tabela é criada para os atributos da classe especializada, gerando assim uma tabela para cada classe. A segunda forma traz para a tabela da superclasse os atributos das classes filhas, gerando assim uma única tabela para cada hierarquia. Já a última forma traz para as classes filhas os atributos da classe pai, gerando assim uma tabela por subclasse. Com o objetivo de gerar um modelo relacional o mais normalizado possível e manter uma coerência clara entre o modelo de classes e o modelo relacional, optamos pelo uso de uma tabela por classe. O código a seguir mostra o descritor XML de configuração do mapeamento objeto-relacional utilizado.

```
<class name="Book" identity-type="datastore">
	<inheritance strategy="new-table"/>
	<field name="author" persistence-modifier="persistent"/>
	<field name="isbn" persistence-modifier="persistent"/>
	<field name="publisher" persistence-modifier="persistent"/>
	<field name="readingLevel" persistence-modifier="persistent"/>
	<field name="reviews" persistence-modifier="persistent" mapped-by="book">
		<collection element-type="model.Review"/>
	</field>
</class>
```

Trecho do arquivo XML responsável pelo mapeamento objeto-relacional da classe Book

Tendo um ambiente configurado, agora é possível desenvolver programas que realizem operações básicas de persistência de objetos como inserção, remoção e recuperação. Os códigos abaixo exemplificam como pode ser feita a persistência de um objeto e a sua posterior recuperação.

```
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory("jpox.properties");
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
	tx.begin();
	Book book = new Book("Harry Potter and the Deathly Hallows","J.K. Rowling", "978-
0545010221", "Arthur A. Levine Books", "Harry Potter book VII", "17.99", "001");
	pm.makePersistent(book);
} finally {
	if (tx.isActive()) { tx.rollback(); }
	pm.close();
}
```

Trecho de código exemplificando como pode ser feita a persistência de um objeto com JDO

Especificação JPA
```
try {
        tx.begin();
        Query q = pm.newQuery("select from model.Book");
        Collection c = (Collection) q.execute();
        Iterator iter = c.iterator();
        while (iter.hasNext()) {
                Book book = (Book)iter.next();
                System.out.println("Name: " + book.getName());
        }
        tx.commit();
} finally {
        if (tx.isActive()) {
                tx.rollback();
        }
        pm.close();
}
```

Trecho de código exemplificando como pode ser feita a recuperação de objetos com JDO

Duas implementações da especificação JPA foram escolhidas para serem cobertas pelo estudo de implementação. O *framework* Hibernate na versão 3.3, em virtude de sua maturidade e reconhecimento no mercado, e o TopLink Essentials na versão 2, implementação de referência da especificação JPA. Dessa forma, o estudo de implementação seguiu a documentação da especificação JPA, e cobriu as duas possibilidades de *frameworks* sobre a mesma aplicação. O objetivo dessa estratégia é garantir que a aplicação não fique presa a particularidades de uma implementação, possibilitando assim uma possível troca de forma menos custosa possível.

A especificação JPA define o mapeamento entre os objetos e o banco de dados relacional a partir de metadados de mapeamento. Esses metadados podem ser expressos na forma de documentos XML, como já vimos no caso da especifcação JDO, ou a partir do uso de Anotações. O uso de anotações permite que as informações a respeito do mapeamento das classes sejam definidas no próprio código da aplicação e possam ser lidas em tempo de execução. O modelo de persistência da aplicação também pode usar de maneira mista as duas formas de declaração. Em casos onde houver conflito entre os mapeamentos, o mapeamento definido por documentos XML sobrescreve o definido por anotações. O código abaixo exemplifica o uso de anotações como forma de declaração do mapeamento objeto-relacional.

```
@Entity
public class Book extends Product {

        @Basic
        private String author = "";
        @Basic
        private String isbn = "";
        @Basic
        private String publisher = "";
        @Basic
        private String readingLevel = "";

        @OneToMany(cascade=CascadeType.ALL, mappedBy="book")
        private Collection<Review> reviews = null;
```

Trecho de código exemplificando o conjunto de anotações responsável pelo mapeamento objeto-relacional

Nesse código podemos notar a presença de algumas das anotações mais importantes para configuração do mapeamento. O @Entity, por exemplo, define que a classe de fato apresenta a propriedade de persistência, ou seja, é uma entidade. A anotação @Basic define que os atributos primitivos devem ser persistidos, sendo essa uma anotação opcional. Assim, todos os atributos de uma entidade por *default* são persistentes. Os atributos que não são persistentes são chamados de transientes, assim esses

atributos devem ser marcados pelo atributo @Transient. Outros atributos que não são traduzidos em tipos primitivos, normalmente representam associações entre classes ou coleções de objetos. Esses relacionamentos são traduzidos para o contexto relacional sobre uma das formas: um-para-um (@OneToOne), um-para-muitos (@OneToMany) ou muitos-para-muitos (@ManyToMany). Assim, a especificação JPA define um conjunto de metadados que cobre essas formas de mapeamento. Os códigos abaixo exemplificam como pode ser feita a persistência de um objeto e a sua posterior recuperação.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(“hibernate”,
                                                    new java.util.HashMap());
EntityManager em = emf.createEntityManager();
try {
       em.getTransaction().begin();
       Book book = new Book("Harry Potter and the Deathly Hallows","J.K. Rowling", "978-
0545010221", "Arthur A. Levine Books", "Harry Potter book VII", "17.99", "001");
       em.persist(book);
       em.getTransaction().commit();
} catch (RuntimeException e) {
       em.getTransaction().rollback();
} finally{
       em.close();
}
```

Trecho de código exemplificando como pode ser feita a persistência de um objeto com JPA

```
try {
       em.getTransaction().begin();
       Query q = em.createQuery("select b from Book b");
       List list = q.getResultList();
       Iterator iterBook = list.iterator();
       while (iterBook.hasNext()) {
              Book book = (Book)iterBook.next();
              System.out.println("Name: " + book.getName());
       }
       em.getTransaction().commit();
} catch (RuntimeException e) {
    em.getTransaction().rollback();
} finally {
       em.close();
}
```

Trecho de código exemplificando como pode ser feita a recuperação de objetos com JPA

Concluindo

A partir dessa análise, pudemos ter uma noção mais específica do funcionamento de cada um dos *frameworks* e especificações discutidos. Com isso, poderemos tomar decisões mais adequadas quando formos confrontados depois de formados com a necessidade de armazenar dados em aplicações reais desenvolvidas na plataforma Java.

Recursos

Leia mais sobre a especificação JDO: http://java.sun.com/jdo

Leia mais sobre a especificação JPA: http://java.sun.com/javaee/technologies/persistence.jsp

Leia mais sobre o *framework* JPOX: http://www.jpox.org

Leia mais sobre o *framework* Hibernate: http://www.hibernate.org

Leia mais sobre o *framework* TopLink Essentials: http://oss.oracle.com/toplink-essentials-jpa.html

Sobre os autores

Hua Lin Costa é aluno de mestrado do Programa de Engenharia de Sistemas e Computação (COPPE/UFRJ) na linha de pesquisa de Engenharia de Software. Possui graduação em Ciência da Computação (2008) pela UFRJ. Atualmente realiza trabalhos acadêmicos junto ao Grupo de Reutilização de Software da COPPE como colaborador no desenvolvimento do ambiente Odyssey e do Projeto Brechó.

Leonardo Murta é sócio da SBC e professor de Engenharia de Software do Instituto de Computação da Universidade Federal Fluminense. Possui graduação em Informática (1999) pela UFRJ e mestrado (2002) e doutorado (2006) em Engenharia de Sistemas e Computação, também pela UFRJ. Suas principais áreas de interesse são gerência de configuração, reutilização e arquiteturas e software. Mais informações podem ser obtidas em http://www.ic.uff.br/~leomurta.

Vanessa Braganholo é sócia da SBC e professora do Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro. Possui graduação em Ciência da Computação (1998) pela Universidade Federal do Rio Grande do Sul e doutorado (2004) em Ciência da Computação também pela UFRGS. Tem atuado em diversos projetos de pesquisa ligados a dados semi-estruturados. Sua principal área de atuação é bancos de dados. Mais informações podem ser obtidas em http://www.dcc.ufrj.br/~braganholo.