Controle Transacional no Hibernate 3 com Anotações

Davi Luan Carneiro

Muito se discute na comunidade Java sobre como se deve fazer controle transacional, e neste artigo mostraremos uma forma simples e funcional de fazê-lo, explicando cada passo.

Introdução

O Hibernate é um framework de mapeamento objeto relacional desenvolvido em Java. Devido às suas grandes potencialidades e facilidade de uso, ele se tornou hoje padrão em termos de persistência com banco de dados, na plataforma Java.

Dentre os seus vários serviços, o Hibernate provê um meio de se controlar transações, através de métodos de suas interfaces **Session** e **Transaction**. A questão, porém, não é simplesmente como fazer o controle transacional, mas sim como fazê-lo de maneira adequada. Existem algumas opções, todas com suas vantagens e desvantagens, tais como:

- Controlar as transações explicitamente dentro das classes Java;
- Utilizar Hibernate em conjunto com EJBs, e configurar o controle transacional com JTA;
- Utilizar o controle de transações do Spring Framework;
- Em ambiente web, criar filters (classes que implementam a interface Filter, da API de Servlets) que comecem uma transação antes do processamento da requisição, e a terminem logo após.

Todas essas opções são válidas, e adequadas em vários casos específicos. Porém queremos, neste artigo, apresentar mais uma: como controlar as transações através de Annotations, um dos novos recursos do Java 5. Não usaremos nenhuma solução pronta, porém nós mesmos vamos implementar uma passo a passo.

Para executar os exemplos deste artigo, é necessário que você tenha o Java 5 instalado, e que faça o download do Hibernate 3, em http://www.hibernate.org. Presumo que você tenha um conhecimento básico do framework.

O que é uma transação?

Transação é uma operação que engloba uma ou mais sub-operações, geralmente relacionadas a bancos de dados, e que possuem estas 4 propriedades fundamentais, conhecidas pelo acrônimo ACID:

→ **Atomicidade:** assegura que a operação não será realizada pela metade. Ela só será confirmada (commit) se todas as sub-operações forem feitas com sucesso. Caso contrário, a operação toda será descartada, e os efeitos das sub-operações até então concluídas serão cancelados (rollback).

→ **Consistência:** uma transação deve sempre zelar para que não produza dados inconsistentes. O próprio conceito de atomicidade deve garantir isso.

→ **Isolamento:** a execução de uma transação não deve interferir na execução de outra. Sem essa propriedade, teríamos grandes chances de uma transação estar manipulando dados inconsistentes.

→ **Durabilidade:** garante que as modificações realizadas na transação serão persistidas, não importando onde. (bancos relacionais, arquivos XML, etc.)

@HibernateTransaction

Utilizaremos esta anotação para marcar os métodos que queremos que sejam transacionais. Consideramos como métodos transacionais aqueles cujo processamento constitui uma única transação com o banco. Confira o código:

```
package br.com.guj.hibernateTransaction;
```

```
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface HibernateTransaction {}
```

HibernateHelper

Esta classe será útil para gerenciar os objetos Session e Transaction, bem como deverá ser usada na confecção dos DAOs (Data Access Object). Ela faz uso de ThreadLocal, garantindo que cada thread possua uma única instância de Session e Transaction, e protegendo esses objetos de qualquer problema de concorrência.

```
package br.com.guj.hibernateTransaction;

import org.hibernate.*;
import org.hibernate.cfg.Configuration;

public class HibernateHelper {
	private static final SessionFactory sessionFactory;
	private static final ThreadLocal sessionThreadLocal = new ThreadLocal();
	private static final ThreadLocal transactionThreadLocal = new ThreadLocal();

	static {
		try {
			sessionFactory = new Configuration().configure()
					.buildSessionFactory();
		} catch (RuntimeException e) {
			e.printStackTrace();
			throw e;
		}
	}

	public static Session currentSession() {
		if (sessionThreadLocal.get() == null) {
			Session session = sessionFactory.openSession();
			sessionThreadLocal.set(session);
		}
		return (Session) sessionThreadLocal.get();
	}

	private static void closeSession() {
		Session session = (Session) sessionThreadLocal.get();
		if (session != null) {
			session.close();
		}
		sessionThreadLocal.set(null);
	}

	static void beginTransaction() {
		if (transactionThreadLocal.get() == null) {
			Transaction transaction = currentSession().beginTransaction();
			transactionThreadLocal.set(transaction);
		}
	}

	static void commitTransaction() {
		Transaction transaction = (Transaction) transactionThreadLocal.get();
		if (transaction != null && !transaction.wasCommitted()
				&& !transaction.wasRolledBack()) {
			transaction.commit();
			transactionThreadLocal.set(null);
		}
		closeSession();
	}

	static void rollbackTransaction() {
		Transaction transaction = (Transaction) transactionThreadLocal.get();
		if (transaction != null && !transaction.wasCommitted()
				&& !transaction.wasRolledBack()) {
			transaction.rollback();
			transactionThreadLocal.set(null);
		}
		closeSession();
	}
}
```

HibernateInterceptor

Esta é uma classe extremamente interessante, sendo o cerne do nosso componente. Ela faz uso de **CGLib**, uma biblioteca que provê mecanismos de manipulação de bytecode, permitindo implementar interfaces dinamicamente, interceptar a chamada de métodos, etc.

Assim, você já deve ter captado o funcionamento do nosso componente: chamando determinado método, interceptamos a sua ação, e verificamos se este método é transacional. Se for, abrimos uma transação, e executamos o método. Se ele for executado com sucesso, damos commit na operação. Porém, se der exceção, fazemos um rollback.

```
package br.com.guj.hibernateTransaction;

import java.lang.reflect.Method;
import net.sf.cglib.proxy.*;

public abstract class HibernateInterceptor implements MethodInterceptor {

	public Object intercept(Object object, Method method, Object[] args, MethodProxy methodProxy)
throws Throwable {
		Object result = null;
		if (isTransactional(object, method)) {
			HibernateHelper.beginTransaction();
		}
		try {
			result = methodProxy.invokeSuper(object, args);
			HibernateHelper.commitTransaction();
		} catch (Exception e) {
			HibernateHelper.rollbackTransaction();
			throw e;
		}
		return result;
	}

	public abstract boolean isTransactional(Object object, Method method) throws Exception;
}
```

Observe que temos uma classe abstrata, onde o método **isTransactional** foi definido como abstrato. Fazemos isso para obter máxima flexibilidade, pois apenas estendendo esta classe e implementando este método, podemos saber se o método é transacional ou não de diversas maneiras, como arquivos XML, arquivos texto, e anotações! (Esta classe é um caso do pattern Template Method).

Já o método **intercept**, proveniente da interface **MethodInterceptor** (CGLib), se encarrega de fazer o controle transacional. Veja que ele usa o **HibernateHelper**, e que o método transacional é explicitamente invocado, através de **methodProxy.invokeSuper()**, e o seu retorno é explicitamente devolvido, em **return result**.

Isso se parece com a solução de Filtros Http, porém com uma vantagem: o controle transacional não fica misturado com a API do Controller (C do padrão MVC), e é mais flexível, pois não se prende ao contexto web. Imagine uma aplicação web com as transações gerenciadas por filters, e imagine que surja a necessidade de implementar uma view em Swing: você terá que reimplementar o controle transacional. Além disso, a solução dos Filters não oferece muita flexibilidade quanto ao tratamento a ser feito quando acontecer o roolback.

Observe também que esta solução é melhor que tratar as transações diretamente nas classes Java. Isso porque você vai evitar muito código repetitivo (try, catch, commit, rollback, etc), e não vai “poluir” inúmeras classes com este tipo de código.

HibernateInterceptorAnnotation

Está é a nossa implementação de HibernateInterceptor. O método isTransactional procura no método a anotação @HibernateTransaction. Se encontrar, retorna true, indicando que o método é transacional. Se não encontrar, retorna false.

```
package br.com.guj.hibernateTransaction;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;

public class HibernateInterceptorAnnotation extends HibernateInterceptor {
	public boolean isTransactional(Object object, Method method) throws Exception {
```

```
                Annotation annotation = method.getAnnotation(HibernateTransaction.class);
                return annotation != null;
        }
    }
```

TransactionClass

Para podermos interceptar os nossos métodos usando CGLib, não podemos criar suas classes através de um construtor. Devemos criá-las a partir do objeto **Enhancer**, fornecido pela API. Assim, implementamos uma classe para nos ajudar na construção de objetos que possuam métodos transacionais.

```
package br.com.guj.hibernateTransaction;

import net.sf.cglib.proxy.Enhancer;

public class TransactionClass {
     public static Object create(Class beanClass, Class interceptorClass) throws Exception {
            HibernateInterceptor interceptor =
     (HibernateInterceptor)interceptorClass.newInstance();
            Object object = Enhancer.create(beanClass, interceptor);
            return object;
     }
}
```

Esta classe possui o método **create**, que recebe dois argumentos do tipo **Class**: o beanClass, que se refere à classe que desejamos criar; e o interceptorClass, que se refere ao tipo de Interceptor usado.
Neste último argumento, passaremos **HibernateInterceptorAnnotation.class**, mas poderíamos também passar **HibernateInterceptorXML.class, HibernateInterceptorTextFile.class**, enfim, qualquer coisa que a sua imaginação permitir.

Recomendo que você não permita que esta classe seja conhecida em vários lugares de sua aplicação. Procure "escondê-la" utilizando o pattern Factory, ou integrá-la com o seu mecanismo de IoC (como Spring Framework ou PicoContainer). Assim, você evita "poluir" suas classes desnecessariamente.

Bem, em termos práticos: se quisermos que determinada classe possua um método transacional, ela deve ser criada por meio de **TransactionClass**.

Aspectos

O nosso exemplo também poderia ser implementado com AOP. Isto seria vantajoso, pois eliminaria a necessidade de se criar os objetos a partir do **Enhancer**. Há vários modos de usar AOP com Java, tais como o AspectJ, JBoss AOP ou Spring AOP.

Exemplo de uso

Agora que já demonstramos como construir o nosso componente, iremos para um exemplo prático: uma transferência bancária.

Essa operação consistirá em duas sub-operações: subtrair a quantia da conta de um dos clientes, e adicionar o mesmo valor na conta do outro. Um débito e um crédito. Observe que temos aqui um típico exemplo da importância de transações. Já pensou se a operação fosse interrompida pela metade? O dinheiro sairia da conta de um, porém não iria para a conta do outro!

Não nos preocuparemos em construir um sistema perfeito, mas faremos o exemplo da maneira mais simples possível, visto que a proposta central é demonstrar o uso da anotação que criamos. Por isso, omitimos a interface **ClienteDAO**, a classe **DAOFactory**, e os arquivos **Cliente.hbm.xml** e **hibernate.cfg.xml**. Estes poderão ser baixados no site do GUJ.

Dessa maneira, segue abaixo a nossa primeira classe, que representa o Cliente do banco.

```
package br.com.guj.hibernateTransaction.example;

public class Cliente {
     private Long id;
     private String nome, senha;
     private Double saldo;
```

```
	public Cliente() {}

	public Cliente(String nome, String senha, Double saldo) {
		this.nome  = nome;
		this.senha = senha;
		this.saldo = saldo;
	}

	public void transfere(Cliente destino, Double valor) throws Exception {
		if (valor <= 0) {
			throw new Exception("Valor inválido: menor que zero");
		}
		if (this.getSaldo() < valor) {
			throw new Exception("Não possui saldo suficiente");
		}

		this.setSaldo(this.getSaldo() - valor);
		destino.setSaldo(destino.getSaldo() + valor);
	}

	//sets e gets omitidos
```

Logo abaixo temos a classe HibernateClienteDAO, implementação de ClienteDAO. Observe que ela deve usar o HibernateHelper.

```
package br.com.guj.hibernateTransaction.example;

import java.io.Serializable;
import java.util.List;
import br.com.guj.hibernateTransaction.HibernateHelper;

public class HibernateClienteDAO implements ClienteDAO {

	public void salvaCliente(Cliente cliente) throws Exception {
		HibernateHelper.currentSession().save(cliente);
	}

	public void removeCliente(Cliente cliente) throws Exception {
		HibernateHelper.currentSession().delete(cliente);
	}

	public void atualizaCliente(Cliente cliente) throws Exception {
		HibernateHelper.currentSession().update(cliente);
	}

	public Cliente getCliente(Serializable id) throws Exception {
		return (Cliente)HibernateHelper.currentSession().get(Cliente.class, id);
	}

	public List<Cliente> listaClientes() throws Exception {
		return HibernateHelper.currentSession().createCriteria(Cliente.class).list();
	}
}
```

Agora, passaremos à classe Banco. Note que ela possui um método **efetuarTransferencia** anotado com **HibernateTransaction**. Portanto, temos é um método transacional.

```
package br.com.guj.hibernateTransaction.example;

import java.io.Serializable;
import br.com.guj.hibernateTransaction.HibernateTransaction;

public class Banco {

	@HibernateTransaction
	public void efetuarTransferencia(Cliente pagador, Cliente favorecido,
			Double valor) throws Exception {
		pagador.transfere(favorecido, valor);

		ClienteDAO dao = DAOFactory.createClienteDAO();
		dao.atualizaCliente(pagador);
		dao.atualizaCliente(favorecido);
	}


	//Por ser um select, não precisa de transação
	public Cliente getCliente(Serializable id) throws Exception {
		return DAOFactory.createClienteDAO().getCliente(id);
```

```
	}
}
```

Por fim, eis a classe Main. Ela pede ao usuário o ID do pagador, do favorecido, e o valor a ser transferido. Veja que o Banco é construído usando o TransactionClass.

```
package br.com.guj.hibernateTransaction.example;

import java.util.Scanner;
import br.com.guj.hibernateTransaction.*;

public class Main {
	public static void main(String[] args) throws Exception {
		try {
			Banco banco = (Banco) TransactionClass.create(Banco.class,
					HibernateInterceptorAnnotation.class);

			Scanner teclado = new Scanner(System.in);

			System.out.println("Qual é o ID do pagador?");
			Cliente pagador = banco.getCliente(new Long(teclado.next()));

			System.out.println("Qual é o ID do favorecido?");
			Cliente favorecido = banco.getCliente(new Long(teclado.next()));

			System.out.println("Qual é o valor a ser transferido?");
			Double valor = new Double(teclado.next());

			//Chamada à operação transacional
			banco.efetuarTransferencia(pagador, favorecido, valor);

			System.out.println("Transferência efetuada com sucesso!");

		} catch (Exception e) {
			System.out.println("Transação não foi efetuada. Confira a stackTrace:");
			e.printStackTrace();
		}
	}
```

Veja que, se tudo der certo, a transação será finalizada e os dados do banco serão atualizados. Se qualquer exceção acontecer, damos rollback.

Conclusão

Neste artigo, procurei demonstrar como fazer um controle transacional inteligente com anotações, para o Hibernate. Esta solução não atenderá a todos os sistemas, mas creio que pode ser útil para muitos deles.

A minha sugestão é que você transforme esta solução (com eventuais extensões) em um componente de software, para que ele possa ser reaproveitado em diversos projetos.

Espero que você tenha aprendido coisas novas!

Referências

hibernate.org

Site oficial do Hibernate

cglib.sourceforge.net

Biblioteca utilizada neste artigo

http://www.argonavis.com.br/cursos/java/j530/j530_12_Transactions.pdf

Apresentação sobre Transações

Davi Luan Carneiro (*daviluan@gmail.com*) é técnico em Informática pelo Colégio Técnico da Unicamp, e graduando em Ciência da Computação pela Universidade Paulista. Programa em Java desde 2004.