Hibernate 3 avançado, boas práticas, padrões e caching

Maurício Linhares de Aragão Junior

*Conheça boas práticas comuns no uso do Hibernate, padrões de projeto relacionados, o desenvolvimento de uma camada de persistência, integração com o **Spring** e aprenda a aumentar ainda mais a velocidade das suas aplicações usando o esquema de caching do framework*

Introdução

O Hibernate é o mais bem sucedido framework de mapeamento objeto/relacional (**ORM – Object/Relational Mapping**) da comunidade Java e o seu sucesso foi tão grande que ele influenciou mudanças drásticas em partes da especificação Java EE (antes J2EE) e ainda gerou uma versão para a plataforma .Net, que ainda está em fase beta. O sucesso do framework vem tanto da facilidade de se começar a trabalhar, quanto da quantidade de opções avançadas que melhoram vários quesitos da aplicação e nós vamos ver um pouco dessas opções e de como usá-las eficientemente.

Isto não é uma introdução

Este artigo não é uma introdução, se você ainda não conhece o Hibernate, veja o artigo "Introdução ao Hibernate 3" (http://www.guj.com.br/java.tutorial.artigo.174.1.guj). Se você já sabe o básico do framework (instalação, configuração, mapeamentos e busca), pode seguir em frente. Se você já leu o primeiro artigo, também não deve ter dificuldades seguindo esta segunda parte, pois o mesmo conjunto de classes é utilizado.

Além das bibliotecas indicadas no primeiro artigo, você vai precisar adicionar o ehcache.jar e as bibliotecas do framework Spring no seu classpath.

Os arquivos de geração do banco de dados estão disponíveis junto com o material em anexo, o banco escolhido para os exemplos foi o MySQL, mas outros bancos relacionais podem ser utilizados, com poucas mudanças do arquivo de criação. O diagrama de classes e de tabelas também está disponível com o material em anexo, além dos arquivos que os geraram.

O modelo das tabelas foi feito com o software DbDesigner e o diagrama de classes com o Jude.

Nos exemplos de código não constam os "imports" nem os "packages" onde se encontram os arquivos para economizar espaço. Nos arquivos deste artigo você pode encontrar os arquivos fonte originais com os "packages" e "imports".

Dicas

Alguns costumes se tornaram comuns na comunidade de usuários, tanto por alguns problemas inerentes da maneira como o framework trabalha, como de entendimentos incorretos do funcionamento de alguns serviços que são providos. Vejamos alguns deles:

Só chame o que você vai precisar

Quando estiver mapeando os seus objetos, defina sempre todos os relacionamentos com o atributo "*lazy*" com o valor "**true**". Isso vai fazer com que sempre que você precisar carregar um objeto, os seus relacionamentos não sejam carregados junto. Se você também precisa receber os objetos relacionados, defina na busca que você os quer, usando a cláusula "left join fetch". No nosso modelo de exemplo, Disciplinas possuem Turmas, vejamos como uma query em HQL faria isso:

Listagem 1 – Exemplo de join em HQL

```
from Disciplina d
       left join fetch d.turmas
```

Quando você não define a os relacionamentos como **lazy="true"** ou usa "left join fetch", o Hibernate vai fazer uma query para os objetos que você queria receber e depois vai fazer mais uma query para cada objeto resultante . Quer dizer, em vez de fazer uma query que traz "N" objetos resultantes, você vai ter (N +1) buscas no banco, o que com certeza vai ser um tremendo gargalo de desempenho na sua aplicação, já que ele vai fazer um **"select"** para cada objeto que foi recebido.

Mas é claro que isso não é uma verdade absoluta, você pode ter que necessariamente carregar algum relacionamento do seu objeto sempre que ele for carregado. Quando isso for necessário, defina o atributo "lazy" como false e o atributo "outer-join" como true.

Use e abuse do suporte a paginação

O Hibernate oferece um esquema portável e interessante de paginação de resultados para todos os bancos que ele suporta oficialmente. Em vez de retornar 1000 objetos de uma só vez para o seu usuário (o que não vai ser nem um pouco interessante pra ele), faça paginação dos resultados, vai ser mais interessante pra ele, que não vai ter condições de interpretar milhares de resultados de uma única vez e também vai melhorar a performance do servidor, que não vai precisar instanciar alguns milhares de objetos a cara requisição.

Vejamos como fazer isso com HQL:

Listagem 2 – Paginando resultados

```
Session session = HibernateUtility.getSession();

Query query = session.createQuery( " from Curso " );
query.setFirstResult( 0 );
query.setMaxResults( 10 );

System.out.print( query.list() );

session.close();
```

No exemplo, criamos a query "from Curso" e informamos ao objeto Query que queremos apenas os 10 primeiros resultados selecionados (na maioria dos bancos, os resultados se iniciam no 0, como em um Array).

Se a busca não é dinâmica, use Named Queries

Quando você sabe que não vai precisar montar a query dinamicamente e já sabe quais vão ser os parâmetros enviados para ela, use uma named query configurada em um dos arquivos de mapeamento do Hibernate. As named queries são compiladas e preparadas no banco de dados assim que o Hibernate se inicializa, o que aumenta a velocidade na qual essas queries é executada (pois nem o banco nem o Hibernate precisam compilar elas outra vez).

Em um arquivo de mapeamento qualquer:

Listagem 3 – Exemplo de named query

```
<query name="listar.usuarios.para.login">
        <![CDATA[
                from Pessoa p
                        where p.nome = :nome and p.email = :email
                        order by p.nome asc
                ]]>
</query>
```

E chamando em código:

Listagem 4 – Chamando uma named query

```
Session session = HibernateUtility.getSession();
Query query = session.getNamedQuery( "listar.usuarios.para.login" );
query.setString("nome", "Mauricio");
query.setString("email", "mauricio.linhares@gmail.com");
```

```
System.out.print( query.list() );
session.close();
```

O código é muito parecido com o anterior, mas em vez de chamar “createQuery()” nós chamamos “getNamedQuery()” passando como parâmetro o nome que foi dado a query no arquivo de mapeamento. A named query pode estar em qualquer dos arquivos de mapeamento do Hibernate, você pode até mesmo criar um arquivo de mapeamento que só contenha named queries. O importante é evitar que queries previsíveis sejam montadas a todo o momento.

Se a busca é dinâmica, use Criteria, não HQL

Você aprendeu HQL, achou uma linguagem interessante e simples de se trabalhar, mas o Hibernate também oferece buscas em uma API orientada a objetos, onde você chama métodos e passa parâmetros para montar uma pesquisa e é especialmente simples para se montar buscas que são construídas dinamicamente.

Muitas vezes por já estar acostumado a trabalhar com HQL, você simplesmente escreve a query em HQL e sai “montando” ela, concatenando Strings e inserindo ou retirando parâmetros conforme eles são descobertos, o que normalmente traz erros que só são descobertos quando a aplicação começa a funcionar e demonstra comportamentos estranhos (ou queries que não funcionam, porque foram montadas incorretamente). Usando a API de Criteria, você pode evitar a maioria desses problemas e ainda ter queries dinâmicas que podem ser montadas facilmente.

As associações em Java não são bidirecionais

Parece estranho ler isso em um primeiro momento, mas isso é uma consideração importante. No último artigo recebi várias dúvidas sobre problemas quando as pessoas tentavam executar o seguinte código:

```
Session session = HibernateUtility.getSession();
HibernateUtility.beginTransaction();

Aluno aluno = new Aluno();
Endereco endereco = new Endereco();

aluno.setNome("Maurício Linhares");
endereco.setCidade("João Pessoa");
aluno.setEndereco(endereco);

session.save(aluno);

HibernateUtility.commitTransaction();
HibernateUtility.closeSession();
```

O erro lançado pelo Hibernate indicava que o objeto “endereco” não tinha como ser inserido porque o valor do objeto “pessoa” relacionado a ele (que no nosso caso é uma subclasse, Aluno) era nulo, mas nós colocamos o objeto “endereco” em “pessoa”, na chamada do método “setEndereco()”, porque isso acontece?

Como eu disse anteriormente, as associações em Java não são bidirecionais automaticamente, você tem que tornar a associação bidirecional **explicitamente** colocando a “pessoa” no “endereco” também, porque o relacionamento entre eles no banco de dados é entre chaves primárias, não pode existir um “endereco” sem uma pessoa relacionada a ele. Para o nosso código funcionar, ele deveria estar assim:

```
Session session = HibernateUtility.getSession();
HibernateUtility.beginTransaction();

Aluno aluno = new Aluno();
Endereco endereco = new Endereco();

aluno.setNome("Maurício Linhares");
endereco.setCidade("João Pessoa");
aluno.setEndereco(endereco);

endereco.setPessoa(aluno);

session.save(aluno);

HibernateUtility.commitTransaction();
HibernateUtility.closeSession();
```

Agora, como nós definimos explicitamente o relacionamento como bidirecional o código funciona normalmente.

"UPDATES" e "DELETES" em massa utilizando o Hibernate

Um dos grandes problemas de se utilizar uma ferramenta de ORM é que você normalmente precisa trazer um objeto para a memória para poder editar ou atualizar algum campo. A coisa fica ainda mais complicada quando você tem que atualizar vários objetos de uma só vez, pois serão ainda mais objetos carregados na memória consumindo os recursos da sua máquina, uma coisa simples que em SQL poderia ser resolvida com um simples e rápido "UPDATE".

Agora com o Hibernate já é possível fazer "UPDATES" e "DELETES" em massa sem ter que carregar os objetos para a memória, a sintaxe dos comandos é simples e muito parecida com as suas contrapartes em SQL, vejamos o comando:

(update | delete) **from** nomeDaClasse **where**

As duas palavras em negrito (from e where) são opcionais, obrigatório mesmo são apenas o comando (update ou delete) e o nome da classe onde esse comando vai ser utilizado. Em um update ou delete no Hibernate você só pode utilizar apenas uma classe e não pode fazer nenhum "join" nem utilizar "apelidos", apenas acessar as propriedades da classe. Vejamos um exemplo de código onde vamos atualizar os nomes de todos os cursos que tenham o nome "DSI":

```
HibernateUtility.beginTransaction();
Session session = HibernateUtility.getSession();

String updateQuery = "update Curso set nome = :novoNome where nome = 'DSI' ";
Query update = session.createQuery( updateQuery );

update.setString( "novoNome", "Desenvolvimento de Software");
int registrosAtualizados = update.executeUpdate();

HibernateUtility.commitTransaction();
HibernateUtility.closeSession();
```

Como você pode ver, montar um "update" é simples e a sintaxe do "delete" é igual (mudando apenas o nome do comando). O método "executeUpdate()" retorna a quantidade de registros atualizadas pelo comando, assim você pode saber se ele surtiu efeito ou não no banco de dados.

Ferramentas

O site http://tools.hibernate.org/ traz as novas ferramentas do Hibernate (para a versão 3.x) para o Eclipse e vários "tasks" do Ant. Os novos plugins se conectam ao banco para testar queries, geram arquivos de mapeamento e até mesmo as classes das tabelas do banco de dados.

Vale a pena fazer os testes com o seu banco de dados, mas não vá com muita sede ao pote, a geração de arquivos de mapeamento e classes Java ainda não é perfeita (e depende muito do suporte a metadata do driver JDBC do seu banco). Quando for utilizálos, faça sempre uma vistoria nos arquivos gerados.

Desenvolvendo uma camada de persistência simples

Uma das partes mais importantes da maioria das aplicações é o acesso as suas fontes de informação (que são muitas vezes bancos de dados relacionais), por isso as camadas de persistência também se configuram como uma parte muito importante de um sistema. Persistência mal feita pode trazer problemas de performance, incompatibilidade entre bancos de dados e dificuldade para a adição de novos recursos.

O Hibernate resolve vários problemas do desenvolvimento desse tipo de camada, com o mapeamento objeto/relacional, esquemas de cache (que nós vamos ver mais a frente) e linguagens que abstraem o

banco de dados que está sendo utilizado, mas ainda existem algumas arestas que devem ser aparadas. Um dos principais é que os objetos que utilizam a camada de persistência (os Actions do seu sistema web ou os formulários da sua aplicação Swing) não deveriam saber que estão utilizando o Hibernate. Idealmente, eles nem deveriam saber que existe um banco de dados relacional "do outro lado", mas isso ainda é um sonho distante nos nossos dias.

Vamos então montar uma camada de persistência com padrões de projeto que podem nos ajudar a manter o código de acesso ao banco longe dos objetos que utilizam a camada, aproveitando ao máximo as possibilidades que o Hibernate nos fornece.

Data Access Object (DAO)

O padrão de projeto Data Access Object (daqui pra frente chamado de DAO) é um velho conhecido da comunidade Java, seu objetivo principal é manter a lógica de acesso a bancos de dados dentro de objetos especializados, que transformam as queries e ResultSets retornados pela API de JDBC em objetos que façam sentido para a aplicação que está fazendo uso dos DAOs.

Se nós não estivéssemos utilizando o Hibernate, poderíamos ter um DAO com um método "Collection listarCursos()" que criaria um Statement, executaria uma query no banco de dados (algo como "select * from Curso"), trataria o ResultSet retornado e transformaria cada linha desse ResultSet em um objeto Curso, que seria adicionado a uma coleção qualquer (uma List, por exemplo) que seria então retornada como resultado da chamada ao método. Mas como estamos usando o Hibernate, todo esse trabalho já é feito pelo próprio framework, o que nos indica que os nossos DAOs vão ser mais simples de serem implementados, como também podem ser mais "poderosos" do que os DAOs comuns, já que o Hibernate nós dá novas escolhas no acesso ao banco.

O primeiro DAO que nós vamos definir é um DAO genérico, que vai lidar com as ações conhecidas como CRUD – Create/Read/Update/Delete (Criar/Ler/Atualizar/Deletar) – que são o básico de qualquer camada de persistência. Aqui nós percebemos uma das primeiras vantagens da camada de persistência montada sobre o Hibernate, porque nós só precisamos de um único DAO genérico para fazer CRUD de todas as classes mapeadas, já que o próprio Hibernate vai cuidar de fazer a persistência no banco de dados. Se estivéssemos usando JDBC puro, provavelmente teríamos um DAO fazendo CRUD para **cada classe mapeada para o banco de dados**, que só no nosso modelo de exemplo geraria **oito classes**, enquanto com o Hibernate apenas uma classe é necessária.

Vejamos a interface que vai definir esse comportamento (e ser implementada por uma classe que use o Hibernate):

Listagem 5 – interface para o DAO base da aplicação

```
public interface GenericDao {

        public void save (Object objeto);

        public void update (Object objeto);

        public void delete(Object objeto);

        public List list (Class clazz);

        public List list (Class clazz, int firstResult, int maxResults);

        public List listByExample(Object example);

        public Object getById(Serializable id, Class clazz);

}
```

Definir essa interface vai nos ajudar a desenvolver várias implementações do DAO (e até mesmo desenvolver uma que não utilize o Hibernate, por exemplo). A interface não demonstra em momento algum qual é o mecanismo de persistência que ela utiliza, os objetos que fizerem uso dela não vão estar presos a o Hibernate nem a nenhuma implementação específica dela.

Os métodos que estão definidos são simples:

- ✓ "save()" – Insere o objeto no banco de dados;
- ✓ "update()" – Atualiza um objeto que já tenha sido inserido no banco de dados;
- ✓ "delete()" – Deleta um objeto do banco de dados;

- ✓ "list()" – Retorna todos os objetos da classe passada como parâmetro que estão no banco de dados, a segunda versão do método faz a mesma coisa, mas com paginação de resultados;
- ✓ "listByExample()" – Retorna todos os objetos que são "parecidos" com o objeto que foi passado como parâmetro;
- ✓ "getById()" – Retorna o objeto que tenha o identificador indicado e pertença a classe passada como parâmetro;

Com a interface definida, podemos começar a pensar em uma possível implementação dela utilizando o Hibernate. Vamos começar por uma classe de apoio que vai nos ajudar a acessar e configurar o Hibernate na nossa aplicação na **listagem 6**:

Listagem 6 – Classe utilitária para configuração e acesso ao Hibernate

```
public class HibernateUtility {

    private static final SessionFactory factory;

    private static final ThreadLocal sessionThread = new ThreadLocal();

    private static final ThreadLocal transactionThread = new ThreadLocal();

    static {
       //Bloco estático que inicializa o Hibernate, escreve o stack trace se houver algum problema e
relança a exceção
       try {
        factory = new Configuration().configure().buildSessionFactory();
       } catch (RuntimeException e) {
              e.printStackTrace();
              throw e;
       }
    }


    public static Session getSession() {
       if ( sessionThread.get() == null ) {
              Session session = factory.openSession();
              sessionThread.set( session );
       }
       return (Session) sessionThread.get();
    }




    public static void closeSession() {
       Session session = (Session) sessionThread.get();
       if ( session != null && session.isOpen() ) {
              sessionThread.set(null);
              session.close();
       }
    }

    public static void beginTransaction() {
       Transaction transaction = getSession().beginTransaction();
       transactionThread.set(transaction);
    }

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

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

}
```

O código da **listagem 6** é a nossa classe utilitária para acessar e configurar o Hibernate no nosso ambiente. Ela se inicia com as declarações de três variáveis constantes, "factory" que vai guardar a SessionFactory do Hibernate e dois objetos ThreadLocal que vão guardar os valores da Session e da Transaction do Hibernate.

Blocos estáticos e exceções

O bloco estático da **listagem 6** pode lançar uma exceção se o Hibernate não for configurado corretamente (ou ocorrer algum erro na sua inicialização), mas o lançamento dessa exceção não vai impedir que a aplicação funcione.

Entretanto, quando a session factory do Hibernate for acessada vai lançar uma exceção avisando que ela não foi configurada corretamente, portanto, preste atenção nos logs que vão ser gerados pela aplicação para não ser pego de surpresa.

Mas por que utilizar ThreadLocal? Os objetos ThreadLocal vão ser utilizados para guardar os valores das sessões e transações do Hibernate para que nós possamos garantir que todos os objetos dentro de uma mesma Thread possam acessar o Hibernate sem problemas de concorrência. Por exemplo, em um ambiente web, onde cada requisição vai necessariamente criar uma nova Thread, nós vamos poder associar essa uma sessão e uma transação nessa Thread no inicio da requisição e esses objetos vão estar disponíveis para todos os objetos criados (ou utilizados) dentro da Thread dessa requisição sem nenhum problema (até mesmo se eles forem criados em outras Threads).

O código da listagem é simples:

getSession() procura uma sessão na Thread corrente, se não houver nenhuma ele cria uma nova sessão, associa ela com a Thread corrente e retorna a sessão;
closeSession() fecha a sessão da Thread corrente;
beginTransaction() inicia uma transação com o banco de dados (veja que o código não procura saber se já existe uma transação, ele sempre abre uma nova);
commitTransaction()testa se existe uma transação na Thread corrente e se é possível fazer o "commit", se estiver tudo certo, ele chama "commit()" na transação;
rollbackTransaction() testa se existe uma transação na Thread corrente e se é possível fazer o "rollback" dela, se for a transação corrente sofre um "rollback";

Agora que já temos uma classe utilitária, podemos desenvolver a nossa primeira implementação do GenericDao utilizando o Hibernate na **Listagem 7**:

Listagem 7 – implementação do GenericDao com o Hibernate

```
public class HibernateGenericDao implements GenericDao {

	public Serializable save(Object objeto) {

		return HibernateUtility.getSession().save(objeto);

	}

	public void update(Object objeto) {

		HibernateUtility.getSession().update(objeto);

	}

	public void delete(Object objeto) {

		HibernateUtility.getSession().delete(objeto);

	}

	public List list(Class clazz) {

		return HibernateUtility.getSession().createCriteria(clazz).list();

	}

	public List list(Class clazz, int firstResult, int maxResults) {
		Criteria criteria =  HibernateUtility.getSession().createCriteria(clazz);

		criteria.setFirstResult(firstResult);
```

```
                criteria.setMaxResults(maxResults);

                return criteria.list();
        }

        public List listByExample(Object example) {
                Criteria criteria = HibernateUtility.getSession().createCriteria( example.getClass()
);

                Example sample = Example.create( example );
                sample.enableLike();
                sample.excludeZeroes();

                criteria.add( sample );

                return criteria.list();
        }

        public Object getById(Serializable id, Class clazz) {
                return HibernateUtility.getSession().get(clazz, id);
        }

}
```

A implementação do DAO também segue a simplicidade que nós esperávamos, a maioria dos métodos conta com apenas uma simples linha de código, apenas dois métodos fugiram a regra, o método “list()” com suporte a paginação e o método “listByExample()” que recebe como parâmetro um objeto de exemplo (que seja um objeto mapeado pelo Hibernate, como um objeto Curso no nosso modelo) e utiliza ele como exemplo para fazer uma busca no banco de dados.

Você já deve ter percebido uma coisa estranha no código, em todos os métodos nós chamamos uma sessão através da classe HibernateUtility, mas em nenhum momento nós iniciamos uma transação para nossas sessões e sem transações não acontecem mudanças no banco de dados. Você não **deveria** colocar código de controle de transações dentro dos seus objetos DAOs, porque pode chegar um momento onde você precise de dois métodos no mesmo DAO executem dentro de uma mesma transação (como fazer uma inserção e logo após uma atualização de algum objeto), como você não quer que apenas uma das ações tenha sucesso sozinha, é necessário manter tudo dentro de uma única transação que **deve ficar fora dos objetos DAO**.

Mas onde chamar as transações?

Existem algumas maneiras de se fazer esse controle, uma maneira comum é adicionar aos objetos DAO métodos que iniciem e terminem uma transação (parecidos com aqueles que temos na classe HibernateUtility), mas esse método traz um outro problema, que é deixar o controle de transações aparente no seu código, pois você vai ter que chamar diretamente os métodos e definir programaticamente quais são as transações, o que termina deixando essa solução até mesmo parecida com o controle de transações dentro do próprio DAO.

Em aplicações web, uma prática comum é criar uma classe filtro (que implementa a interface javax.servlet.Filter) , essa classe vai abrir uma sessão e iniciar uma transação com o banco de dados para que todas as chamadas na requisição acessem essa sessão e essa transação, além disso, essa classe também vai ser responsável por fazer o “rollback()” da transação se alguma exceção for lançada e no fim vai fechar a sessão em todos os casos, liberando o código da aplicação de se preocupar com isso.

Vejamos na **listagem 8** como esse filtro poderia ser implementado:

Listagem 8 – filtro para uso do Hibernate em aplicações web

```
public class FiltroDoHibernate implements Filter {

        public void init(FilterConfig config) throws ServletException {

        }

        public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {

                HibernateUtility.beginTransaction();

                try {
                        chain.doFilter(request, response);
                        HibernateUtility.commitTransaction();
```

```
                } catch (HibernateException exception) {
                        exception.printStackTrace();
                        HibernateUtility.rollbackTransaction();

                } finally {
                        HibernateUtility.closeSession();
                }

        }

        public void destroy() {

        }

}
```

Mas essa implementação também apresenta problemas, porque ela pressupõe automaticamente que cada requisição feita ao sistema é **apenas uma transação** e em algum momento pode ser necessário o uso de várias transações dentro de uma mesma requisição (mesmo que na maioria dos casos seja mesmo apenas uma requisição por transação). Outro problema é que ela provavelmente vai iniciar transações até mesmo quando elas não são necessárias, o que pode gerar um gasto desnecessário de recursos da aplicação e do banco de dados.

Filtros e versões de containers

Para utilizar filtros em aplicações web em Java você necessita de um container de servlets (um servidor web Java) que tenha suporte a especificação Servlet 2.3 ou superior (como por exemplo, o Tomcat 4.x ou 5.x).

Se você não está em um ambiente web ou o filtro não resolve o seu problema por causa de transações complexas demais, ainda existe uma solução para o seu problema que é ainda melhor do que as anteriores, além de deixar o código muito mais simples de ser mantido, que é utilizar o framework Spring para controlar as suas transações do Hibernate.

E os outros DAOs?

Com esse DAO genérico nós já diminuímos em muito a quantidade de código envolvida na camada de persistência, mas sendo genérico, ele também não pode ter comportamentos que só funcionem em classes específicas. Para essas classes específicas devem ser criados outros DAOs específicos, vejamos um exemplo com um DAO específico para cursos (ele tem apenas um método, mas poderiam haver mais se fosse necessário) nas listagens 9 e 10:

Listagem 9 – Interface CursoDao

```
public interface CursoDao extends GenericDao {

        public List listarCursosPorDisciplina (Disciplina disciplina);

}
```

Listagem 10 – Implementação da interface CursoDAO

```
public class HibernateCursoDao extends HibernateGenericDao implements CursoDao {

        public List listarCursosPorDisciplina(Disciplina disciplina) {

                Session session = HibernateUtility.getSession();
                Query query =  session.getNamedQuery("listar.cursos.por.disciplina");
                query.setInteger( "id", disciplina.getId() );

                return query.list();
        }

}
```

Um ponto importante a ser percebido na nossa implementação é que a interface CursoDao extende a interface GenericDao forçando a implementação dessa interface pela classe que implementar CursoDao. Isso não é uma obrigação, você pode simplesmente retirar essa necessidade, isso é apenas uma

conveniência, pois eu posso simplesmente usar CursoDao quando estiver trabalhando com Cursos, em vez de ter que trabalhar com GenericDao e também com CursoDao.

Você não precisa criar DAOs para cada objeto do seu modelo, você pode ter DAOs de "módulos", que reúnem comportamentos de módulos específicos do seu sistema, trabalhando com vários objetos diferentes. No nosso exemplo poderia existir um DAO específico para o módulo de inscrição nas turmas, que englobaria métodos para lidar com Turmas, Alunos, Professores e Disciplinas. Quando estiver montando a sua aplicação, avalie a quantidade de métodos e o tamanho dos DAOs, se eles ficarem grandes demais, está na hora de você começar a dividir as responsabilidades.

O que é o framework Spring?

O Spring é um framework de inversão de controle e de provimento de serviços para aplicações Java. Ele torna o desenvolvimento de aplicações Java EE e até mesmo Java SE mais simples abstraindo e simplificando acesso a recursos avançados (e complexos) como controle de transações, disponibilização de objetos remotos e outras funcionalidades.

Ele também contém várias classes de apoio para resolver problemas comuns, inclusive classes para facilitar o uso de frameworks de mapeamento objeto/relacional como o Hibernate, que nós vamos utilizar nesse artigo.

Você pode encontrar material de introdução ao Spring na edição número 13 da revista MundoJava ou nesse tutorial do TheServerSide.com.

Para fazer os exemplos dessa parte funcionarem, além dos .JAR do Hibernate, você precisa colocar os .JAR do Spring e do DBCP (que vem na distribuição do Spring) no seu classpath.

Desenvolvendo uma camada de persistência com Spring e Hibernate

Vejamos agora como implementar a nossa camada de persistência utilizando o Spring junto do Hibernate. Utilizando o Spring, nós vamos nos livrar de vez do controle de transações em código e tudo vai ser feito de forma declarativa, no arquivo de configuração do Spring. E além disso, ainda vamos utilizar como base as classes de suporte a acesso a banco de dados do Spring, que vão suplantar a nossa classe HibernateUtility e deixar os nossos DAOs ainda mais simples. Antes de passar para a integração com o Hibernate, vamos entender como o Spring faz o controle de transação das aplicações.

Nível de isolamento e comportamento de propagação de transações

O **nível de isolamento** de uma transação simboliza o quanto ela é segura com relação a outros agentes que estejam acessando (ou tentando acessar) as informações contidas dentro daquela transação. É no nível de isolamento que você vai definir o quanto esses agentes externos podem influenciar na sua transação.

Você poderia definir, por exemplo, que ninguém pode acessar ou atualizar essas informações no banco de dados enquanto você estiver com elas dentro de uma transação, mas isso vai gastar muitos recursos e pode diminuir e muito a performance da sua aplicação, por outro lado, se você deixar que outras pessoas vejam e atualizem as informações poderia inserir dados "falsos" no sistema, o que poderia acarretar problemas maiores ainda. Então você deve pensar bem antes de escolher o nível de isolamento de uma transação.

O **comportamento de propagação** define como ela deve adquirir e lidar com as suas transações, você poderia por exemplo ter uma ação que nunca deveria executar dentro de uma transação e outra que só deveria executar se já houvesse uma transação corrente, nunca iniciando uma nova transação.

No Spring você pode trabalhar dois tipos diferentes de gerência de transações:

- ✓ Transações declarativas: São as transações que ficam definidas em arquivos de configuração. São o tipo mais simples e mais utilizado, porque deixam o seu código livre de transações e ainda podem ser trocadas de maneira simples, normalmente apenas editando um arquivo de configuração. Transações definidas como atributos também podem ser consideradas declarativas, mas elas são mais complexas porque exigem a compilação da classes.
- ✓ Transações programáticas: São as transações que ficam definidas dentro do seu código. Não são uma boa escolha porque se houver necessidade de mudar alguma coisa você vai ter que mexer no seu código, mas podem existir casos onde ela é necessária (transações programáticas com o Spring não vão ser vistas nesse artigo).

A interface base de suporte a transações no Spring é PlataformTransactionManager, cada plataforma (como JTA, JDBC, JMS e outras) tem uma implementação específica dessa interface, que reúne os serviços essenciais para a gerência de transações em uma aplicação. As informações de uma transação (como a propagação, o tempo de espera e o nível de isolamento) são encapsulados em uma classe que implementa a interface TransactionDefinition. Também é na interface TransactionDefinition que nós vamos encontrar as constantes que simbolizam o nível de isolamento e a propagação das transações vejamos essas constantes:

Nível de isolamento	Descrição
TransactionDefinition.ISOLATION_DEFAULT	É o nível geral definido no PlataformTransactionManager utilizado.
TransactionDefinition.ISOLATION_READ_UNCOMMITED	Nesse nível as transações podem ler informações que ainda não receberam um “commit”. Esse nível não é indicado porque você pode estar trabalhando com informações que não são confiáveis.
TransactionDefinition.ISOLATION_READ_COMMITED	Esse nível garante que você só vai ler informações que receberam um “commit”, é o nível mais comum de ser utilizado.
TransactionDefinition.ISOLATION_REPEATABLE_READ	Esse nível garante que você pode selecionar a mesma informação mais uma vez, mesmo que ela tenha sido alterada durante uma outra transação.
TransactionDefinition.ISOLATION_SERIALIZABLE	Esse nível garante que todas as transações são executadas sequencialmente, uma após a outra, sem que haja nenhuma influência entre as informações que eles lidam. É o modo mais confiável mas também é o que exige mais recursos da aplicação.

Comportamento de propagação	Descrição
TransactionDefinition.PROPAGATION_REQUIRED	Indica que o uso de uma transação é obrigatório. Se já houver uma transação ela é utilizada, se não houver nenhuma uma nova é criada.
TransactionDefinition.PROPAGATION_SUPPORTS	Indica que se houver uma transação ela é utilizada, se não houver nenhuma a ação é executada de forma não-transacional.
TransactionDefinition.PROPAGATION_MANDATORY	Indica que a existência de uma transação é obrigatória. Se não houver uma transação acontecendo uma exceção é lançada.
TransactionDefinition.PROPAGATION_NOT_SUPPORTED	Indica que a ação deve ser executada sempre fora de uma transação. Se já houver uma transação ela é suspendida e a ação é

	executada e forma não-transacional.
TransactionDefinition.PROPAGATION_REQUIRES_NEW	Indica que uma nova transação sempre vai ser iniciada. Se já houver uma transação ocorrendo ela é suspensa.
TransactionDefinition.PROPAGATION_NEVER	Indica que nunca vai ser executado dentro de transações. Se houver uma transação corrente uma exceção é lançada.
TransactionDefinition.PROPAGATION_NESTED	Executa em transações aninhadas (uma transação acontecendo dentro de outra transação). Se não houver nenhuma transação ocorrendo se comporta de forma igual a PROPAGATION_REQUIRED

Agora que você já entendeu a base do controle de transações do Spring, vamos voltar para a nossa camada de persistência, implementando a interface GenericDao com o Spring na listagem 11:

Listagem 11 – Implementação da interface GenericDao com o Spring e Hibernate

```
public class HibernateGenericDao extends HibernateDaoSupport implements
                GenericDao {

        public Serializable save(Object objeto) {
                return this.getHibernateTemplate().save(objeto);
        }

        public void update(Object objeto) {
                this.getHibernateTemplate().update(objeto);
        }

        public void delete(Object objeto) {
                this.getHibernateTemplate().delete(objeto);
        }

        public List list(Class clazz) {
                return this.getHibernateTemplate().loadAll(clazz);
        }



        public List list(Class clazz, int firstResult, int maxResults) {

                return
                   this.getHibernateTemplate()
                        .executeFind( new CriteriaListCallback(clazz, firstResult, maxResults) );

        }

        public List listByExample(Object example) {
                return this.getHibernateTemplate().findByExample(example);
        }

        public Object getById(Serializable id, Class clazz) {
                return this.getHibernateTemplate().get(clazz, id);
        }

        private class CriteriaListCallback implements HibernateCallback {

                private Class clazz;

                private Integer inicio;

                private Integer quantidade;

                public CriteriaListCallback(Class clazz, Integer inicio,
                                Integer quantidade) {
                        this.clazz = clazz;
                        this.inicio = inicio;
                        this.quantidade = quantidade;
                }

                public Object doInHibernate(Session session) throws HibernateException,
```

```
                        SQLException {

                    Criteria criteria = session.createCriteria(clazz);

                    criteria.setFirstResult(inicio);
                    criteria.setMaxResults(quantidade);

                    return criteria.list();
                }

        }

}
```

A nova implementação da interface GenericDao também é quase tão simples quanto a anterior, utilizando a classe utilitária HibernateUtility, mas ela tem algumas diferenças muito importantes. Uma das principais é que ela extende uma classe do Spring, HibernateDaoSupport, que fornece vários métodos e objetos utilitários para se trabalhar com o Hibernate.

Uma das principais características da classe HibernateDaoSupport é o objeto HibernateTemplate, que pode ser acessado através to método getHibernateTemplate(), ele oferece atalhos para vários métodos da Session do Hibernate, sem acessar a Session diretamente, como save(), update() e delete(). O HibernateTemplate também executa objetos que implementem a interface HibernateCallback, que serve para se definir lógicas mais complexas que necessitem realmente de acesso a um objeto Session.

No nosso exemplo, criamos um objeto que implementa a interface HibernateCallback para executar a listagem de objetos com paginação. Veja que o objeto CriteriaListCallback tem três propriedades, que são a classe que vai ser listada, o resultado inicial e a quantidade máxima de resultado, usando essa classe nós acessamos um objeto Session sem correr nenhum risco de atrapalhar o controle de transações, já que ele vai ser chamado através do objeto HibernateCallback. Sempre que você necessitar de uma lógica complexa nas suas queries do Hibernate, monte um objeto que implemente a interface HibernateCallback para fazer o serviço.

Agora que nós já temos uma implementação com o Hibernate, vamos ver como ficaria a configuração do Spring para controlar a transação do nosso DAO:

Listagem 12 – Configuração do Spring

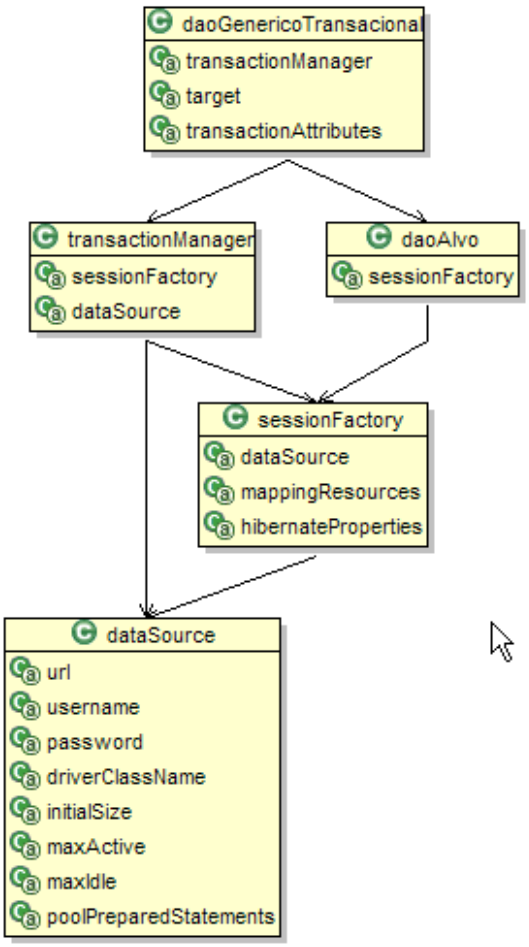
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource">
          <ref local="dataSource"/>
        </property>
        <property name="mappingResources">
          <value>
                br/edu/cefetpb/Curso.hbm.xml,
                br/edu/cefetpb/Disciplina.hbm.xml,
                br/edu/cefetpb/Turma.hbm.xml,
                br/edu/cefetpb/Pessoa.hbm.xml,
                br/edu/cefetpb/Aluno.hbm.xml,
                br/edu/cefetpb/Professor.hbm.xml,
                br/edu/cefetpb/Endereco.hbm.xml
          </value>
        </property>
        <property name="hibernateProperties">
          <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.MySQLDialect
```

```
            </prop>
            <prop key="show_sql">true</prop>
            <prop key="hibernate.generate_statistics">true</prop>
            <prop key="hibernate.use_sql_comments">true</prop>
          </props>
        </property>
    </bean>

    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
      <property name="url">
        <value>jdbc:mysql://localhost/hibernate?autoReconnect=true</value>
      </property>
      <property name="username">
        <value>root</value>
      </property>
      <property name="password">
        <value></value>
      </property>
      <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
      </property>
      <property name="initialSize">
        <value>5</value>
      </property>
      <property name="maxActive">
        <value>20</value>
      </property>
      <property name="maxIdle">
        <value>5</value>
      </property>
      <property name="poolPreparedStatements">
        <value>true</value>
      </property>
    </bean>

    <bean id="daoAlvo"
        class="br.edu.cefetpb.spring.HibernateGenericDao">
        <property name="sessionFactory">
          <ref local="sessionFactory"/>
        </property>
    </bean>


    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory">
          <ref local="sessionFactory"/>
        </property>
        <property name="dataSource">
          <ref local="dataSource"/>
        </property>
    </bean>

    <bean id="daoGenericoTransacional"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
      <property name="transactionManager">
        <ref local="transactionManager"/>
      </property>
      <property name="target">
        <ref local="daoAlvo"/>
```

```
      </property>
      <property name="transactionAttributes">
        <props>
          <prop key="delete*">
            PROPAGATION_REQUIRED, ISOLATION_READ_COMMITTED
          </prop>
          <prop key="save*">
            PROPAGATION_REQUIRED, ISOLATION_READ_COMMITTED
          </prop>
          <prop key="update*">
            PROPAGATION_REQUIRED, ISOLATION_READ_COMMITTED
          </prop>
          <prop key="get*">
            PROPAGATION_SUPPORTS, readOnly
          </prop>
          <prop key="list*">
            PROPAGATION_SUPPORTS, readOnly
          </prop>
        </props>
      </property>
    </bean>

</beans>
```

Na nossa configuração do Spring, foram definidos 5 objetos, que vão ser utilizados para o nosso controle de transações, vejamos um gráfico de relacionamentos entre eles:

Imagem 1 – Beans do Spring



Agora vamos entender o que é cada um desses objetos no arquivo de configuração do Spring:

- ✓ **dataSource**: é o objeto que faz conexão com o banco e é utilizado pela SessionFactory do Hibernate para criar novas conexões JDBC, no nosso exemplo foi escolhido o data source do projeto Jakarta Commons DBCP (o mesmo data source utilizado pelo Tomcat);

- ✓ **sessionFactory:** é a SessionFactory do Hibernate, mas configurada por uma classe do Spring, a classe org.springframework.orm.hibernate3.LocalSessionFactoryBean. Você pode até utilizar a própria SessionFactory do Hibernate, mas vai perder muito das facilidades que o Spring lhe fornece. Além das configurações normais do Hibernate, esse objeto também recebe um dataSource para obter conexões JDBC.

- ✓ **transactionManager:** aqui nós já estamos chegando no controle de transação. O transactionManager escolhido foi o HibernateTransactionManager, que é a implementação da interface PlatformTransactionManager (citada acima) que lida com as transações do Hibernate. Esse recebe como dependências a sessionFactory do Hibernate e (opcionalmente) o dataSource que está sendo utilizado.

- ✓ **daoAlvo:** é a nossa implementação do Spring para o GenericDao, como dependência ele recebe a sessionFactory do Hibernate. Não é esse o objeto que nós vamos utilizar, porque ele ainda não é transacional, ele só está aqui para ser "transformado" em um objeto transacional.

Por último temos o **daoGenericoTransacional**. Esse é o mais complicado dos nossos objetos, ele é um objeto "proxy" (um objeto que serve de acesso para outro objeto ou alguma outra coisa). É esse objeto proxy que faz todo o controle de transações no nosso objeto daoAlvo e é esse objeto que nós vamos utilizar na nossa aplicação.

Ele encapsula o nosso objeto DAO e usa o transactionManager para montar as transações conforme ele está configurado na propriedade "transactionAtributes". Essa propriedade é um objeto Properties onde as chaves são os nomes dos métodos e os valores são as definições de isolamento e do comportamento de propagação das transações, como por exemplo na propriedade seguinte:

```
<prop key="list*">
  PROPAGATION_SUPPORTS, readOnly
</prop>
```

Esse valor do objeto Properties indica que todos os métodos que começarem com "list" no nome usam a propagação PROPAGATION_SUPPORTS e o nível de isolamento readOnly (não vimos esse isolamento anteriormente, mas ele indica que é "somente-leitura", não existe mudança no banco de dados).

Como o objeto **daoGenericoTransacional** é um "proxy", a nossa aplicação nunca vai saber onde fica o código de gerência de transações, porque para ela o objeto que está sendo utilizado é um GenericDao, ela não sabe que está acessando um objeto "proxy" que faz todo o controle de transações. Vejamos um código que faz uso desse objeto na listagem 12:

Listagem 12 – Teste do JUnit que demonstra o uso do proxy

```
public class SpringTestExemplos extends TestCase {

	private ApplicationContext context;


	protected void setUp() throws Exception {
		super.setUp();
		context = new ClassPathXmlApplicationContext("applicationContext.xml");
	}

	public void testInserirCurso() {
		GenericDao dao = (GenericDao) context.getBean("daoGenericoTransacional");

		Integer quantidadeAnterior = dao.list(Curso.class).size();

		Curso curso = new Curso();

		curso.setNome("DSI");
		curso.setDescricao("Desenvolvimento de Software Para internet");

		dao.save(curso);

		if ( quantidadeAnterior.equals( dao.list(Curso.class) ) ) {
			this.fail("A quantidade de cursos deveria ter sido aumentada");
		}

	}


}
```

Como você percebeu, não existe nada de estranho no código, apenas pegamos o bean do Spring e chamamos os métodos nele, em nenhum momento o código cliente (o do teste do JUnit) soube que estava trabalhando com um objeto "proxy", nem mesmo com um banco de dados e é exatamente esse nível de abstração que nós estamos procurando.

A gerência de transações continua no DAO?

No nosso exemplo, o DAO se tornou um objeto transacional, mas nada impede que **qualquer outro objeto** torne-se transacional, o DAO foi escolhido apenas por simplicidade. Você poderia ter, por exemplo, um objeto **GerenciadorDeTransferências** com um método parecido com o da listagem 13:

Listagem 13 – método de transferência de contas

```
public void transferir (ContaCorrente sacado, ContaCorrente beneficiado, Double valor) {
	sacado.remover(valor);
	beneficiado.adicionar(valor);
	this.getGenericDao().update(sacado);
	this.getGenericDao().update(beneficiado);

}
```

E é claro que você quer que ele execute todo dentro de **uma única transação**, porque senão você pode retirar dinheiro do sacado e não depositar no beneficiado, o que seria um grande problema pro seu banco e pro seu emprego também!

Bastaria configurar o objeto **GerenciadorDeTransfências** como um "proxy", do mesmo jeito que foi configurado o **daoGenericoTransacional**, como nós vamos ver na listagem 14:

Listagem 14 – parte da configuração do Spring para o GerenciadorDeTransferências transacional

```
      <bean id="gerenciadorDeTransferenciasAlvo"
                  class="br.edu.cefetpb.banco.GerenciadorDeTransferencias">
            <property name="genericDao">
                  <ref bean="daoAlvo"/>
            </property>
      </bean>

    <bean id="gerenciadorDeTransferencias"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
      <property name="transactionManager">
        <ref local="transactionManager"/>
      </property>
      <property name="target">
        <ref local="gerenciadorDeTransferenciasAlvo"/>
      </property>
      <property name="transactionAttributes">
        <props>
          <prop key="transferir*">
            PROPAGATION_REQUIRED, ISOLATION_SERIALIZABLE
          </prop>
        </props>
      </property>
    </bean>
```

Como você pode perceber, o caminho foi o mesmo, só que em vez de criar um proxy com base no DAO, nós criamos um proxy com base no GerenciadorDeTransferencias, além disso, nós colocamos o DAO **que não é transacional** no gerenciador, porque se nós colocássemos o transacional poderíamos perder informações (e você perder o emprego =]).

No fim, você deve escolher onde o seu controle de transações deve ficar e você tem toda a liberdade pra isso utilizando o Spring, pois apenas editando o arquivo de configuração dele você pode mudar completamente o controle de transações da sua aplicação, sem ter que recompilar nem uma linha de código.

Spring em aplicações web

Trabalhando com aplicações web junto com o esquema de gerência de transações do Spring, você tem ainda um filtro que mantém sempre uma sessão aberta para cada requisição as suas páginas JSP. Com essa sessão aberta você evita problemas de carregamento de objetos e vai ter sempre uma sessão aberta para fazer o "lazy-load" dos seus objetos, evitando assim algumas das mais chatas exceções lançadas pelo Hibernate, como **LazyInitializationException** (que acontece quando você tenta acessar uma coleção ou objeto não carregado sem uma sessão aberta) e **NonUniqueObjectException** (que acontece quando você adiciona o mesmo objeto duas vezes a mesma Session).

O filtro é o **org.springframework.orm.hibernate3.support.OpenSessionInViewFilter**, basta adicionar ele com filtro no seu web.xml para todas as páginas que precisem de uma sessão do Hibernate.

Cache de objetos no Hibernate

Sempre que você encontra alguém falando mal de frameworks que facilitam trabalhar com bancos de dados (especialmente frameworks de mapeamento objeto/relacional (ORM – Object/Relational Mapping) como o Hibernate) uma das principais críticas é a performance da aplicação que usa essas ferramentas.

Realmente, a performance que pode ser conseguida com acesso direto via JDBC é realmente maior se compararmos diretamente com a performance de uma aplicação que use um desses mecanismos.

Mas é claro que as ferramentas de ORM se preocupam com performance, uma das principais demonstrações disso é que praticamente todas elas tem um esquema de cache de objetos, para evitar idas desnecessárias ao banco de dados. Os caches de objetos, quando bem montados, podem tornar a aplicação ainda mais rápida do que a mesma aplicação feita com JDBC, que dificilmente vai ser capaz de lidar com caches de objetos de maneira satisfatória (a não ser que além de fazer a camada de persistência "na mão" você ainda vá montar um esquema de cache, quando você terminar tudo, será que ainda vai ter prazo pra fazer a aplicação?).

O Hibernate tem um sistema de cache automático e não-desligável, que é o "cache de primeiro nível". O cache de primeiro nível é um cache relacionado a uma única sessão (Session) do Hibernate, esse cache garante que você vai acessar sempre os mesmos objetos dentro de uma única sessão. Por exemplo, se você fizer um "load()" em uma sessão passando o mesmo identificador e a mesma classe duas vezes o Hibernate deve retornar o **mesmo objeto,** isso garante que você não vai trabalhar com objetos que tenham valores inválidos dentro de uma mesma sessão. Esse cache não tem nenhuma preocupação com performance, o seu objetivo é garantir a integridade das informações dentro de uma mesma Session.

O outro sistema de cache, que é opcional e é o que nós vamos ver aqui, é o "cache de segundo nível", que é o cache de objetos do Hibernate. O cache de segundo nível pode ser em cluster ou por SessionFactory (ou por "processo"). O que nós vamos ver aqui é o cache por SessionFactory (também existe o cache de queries, mas não vamos vê-lo neste artigo).

Antes de começar a planejar o cache da sua aplicação você deve levar algumas coisas em consideração e uma das mais importantes é que se houverem outras aplicações fazendo inserções e atualizações no mesmo banco de dados que o Hibernate está conectado, é melhor nem utilizar o cache, porque a outra aplicação pode atualizar informações que podem não ser percebidas pelo Hibernate por elas estarem em cache. Outro porém é se a sua aplicação tem mais inserções e atualizações do que leituras, nesse caso o cache pode até mesmo diminuir a performance da aplicação.

Pense bem antes de resolver que a sua aplicação precisa de um cache e escolha os objetos que devem ficar no cache, algumas indicações sobre bons candidatos:

- ✓ Objetos que contém apenas "meta-informação", como por exemplo um Caderno em um jornal, os cadernos normalmente simplesmente indicam qual o "tipo" do assunto abordado;
- ✓ Objetos pouco atualizados e lidos com muita freqüência. Mais um exemplo "jornalístico" são as notícias, você não vai ver uma notícia ser atualizada várias vezes depois que ela foi inserida (a não ser em casos bem específicos);
- ✓ Objetos onde a "incoerência" de informações não vai causar danos ao funcionamento da aplicação;

Se você tem objetos que são atualizados com muita freqüência ou onde é muito importante que eles estejam sempre atualizados, não os coloque no cache, isso pode lhe dar **muita dor de cabeça**.

Esse cache mantém os objetos utilizados na memória ou em disco e é extremamente simples de ser utilizado. No seu mapeamento, basta adicionar um novo nó, o <cache/>, como no nosso exemplo na listagem 15:

Listagem 15 – Exemplo de mapeamento com configuração de cache

```
<class name="Curso">

      <cache
            usage="read-write"
            region="curso"/>

      <id name="id">
            <generator class="increment"/>
      </id>

      <property name="nome"/>
      <property name="descricao"/>
```

```
            <set name="disciplinas"
                    inverse="true"
                    cascade="save-update">

                  <key column="Curso_id"/>
                  <one-to-many class="Disciplina"/>

             </set>

      </class>
```

A configuração do cache é extremamente simples, você indica o modo de utilização (usage) e a região onde os objetos vão ser encontrados (region). Uma região é um espaço de cache que vai ser utilizado para guardar os objetos, você pode ter todos os seus objetos em uma única região ou pode ter uma região diferente para cada objeto que vai para o cache. O melhor é trabalhar com uma região para cara objeto, pois assim se você precisar retirar todos os objetos de um certo tipo do seu cache, pode simplesmente fazer um "evict()" na região onde ele está e todos os objetos vão ser retirados do cache.

Modos de utilização do cache do Hibernate

Os modos de utilização são a maneira pela qual o Hibernate vai controlar o acesso e o comportamento dos objetos que estão no cache, eles são definidos no atributo "usage" do elemento <cache/> nos arquivos de mapeamento do Hibernate. Existem quatro tipos de modos de utilização:

- read-only – é utilizado quando a aplicação não atualiza as informações, ela apenas as lê;
- read-write – deve ser utilizado quando a aplicação faz atualizações nos objetos que estão no cache, mas não deve ser utilizado se o nível de isolamento das transações for ISOLATION_SERIALIZABLE;
- nonstrict-read-write – deve ser utilizado quando a quantidade de atualizações dos objetos é pequena e quando é pouco provável que duas transações vão tentar atualizar o mesmo objeto;
- transactional – só pode ser utilizado em um ambiente com suporte a JTA (como um servidor de aplicações completo como o JBoss). Esse tipo de cache é totalmente transacional e pode ser utilizado com qualquer nível de isolamento das transações. O único cache suportado oficialmente pelo Hibernate 3.x é o JBoss TreeCache;

Na maioria das vezes você vai trabalhar com os modos read-write ou nonstrict-read-write, mas antes de definir isso, avalie o comportamento de transações do seu sistema e as possibilidades de perda ou uso de informações incorretas. Se você encontrar problemas, é melhor não usar o cache.

Para habilitar o cache na nossa aplicação, nós precisamos escolher uma implementação da interface CacheProvider e adicionar ela a configuração do Hibernate. Nós vamos utilizar a implementação EhCacheProvider que é uma implementação não-clusterizável da interface CacheProvider para o Hibernate (isso significa que ela funciona como cache de processo, mas não deveria ser utilizada em um ambiente de cluster). Para indicar isso, vamos adicionar uma linha na nossa configuração do Hibernate no Spring (ela também pode ser adicionada da mesma forma a configuração normal do Hibernate):

```
<prop key="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</prop>
```

Essa configuração indica que nós vamos utilzar o EhCache como implementação de cache para a nossa aplicação. Agora que já indicamos qual é a implementação, temos que definir as configurações do EhCache para a nossa aplicação na listagem 16 (o arquivo deve se chamar ehcache.xml e estar na raiz do classpath da sua aplicação):

Listagem 16 – Exemplo de configuração do EhCache

```
<ehcache>
```

```
    <diskStore path="user.home"/>

    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="true"
        />

    <cache name="curso"
        maxElementsInMemory="50"
        eternal="true"
        timeToIdleSeconds="0"
        timeToLiveSeconds="0"
        overflowToDisk="false"
        />

</ehcache>
```

O arquivo de configuração do EhCache também é simples, o primeiro elemento, <diskStore/> contém a localização de onde deve ficar o arquivo que contém o cache (se algum objeto não for ser guardado na memória). Você pode indicar um diretório ou usar os "apelidos":

- ✓ user.home – indica o diretório raiz do usuário;
- ✓ user.dir – indica o diretório corrente para aplicação;
- ✓ java.io.tmpdir – indica o diretório temporário pra a máquina virtual;

O segundo elemento, <defaultCache/>, indica as configurações "gerais" para o cache na aplicação, ele só vai ser utilizado se o objeto enviado para o cache não estiver relacionado a nenhuma região. Os atributos são:

- ✓ maxElementsInMemory – a quantidade máxima de objetos que vão ficar em memória, além disso, o cache começa a enviar os objetos para o disco (apenas se o valor de overflowToDisk for true);
- ✓ eternal – indica se o conjunto é "eterno". Conjuntos eternos não levam e condição o tempo de vida nem o tempo se uso. Esse modo é normalmente utilizado para objetos que tenham a utilização no cache de "read-write" ou "read-only".
- ✓ timeToIdleSecons – indica a quantidade de segundos que um objeto pode passar no cache sem ser utilzado, após o limite ele é retirado;
- ✓ timeToLiveSeconds – indica a quantidade de segundos que um objeto pode passar no cache, após o limite ele é retirado;
- ✓ overflowToDisk – indica se os objetos devem ou não ser enviados para um arquivo no disco quando a quantidade máxima de objetos em memória for alcançada;

O terceiro (e último) elemento é o <cache/>, que contém as informações específicas de cada região de cache. Além dos atributos de <defaultCache/>, esse elemento também contém o atributo "name", que é o nome da região que identifica esse cache (no nosso exemplo nós demos o nome "curso" que é o mesmo nome definido na região do mapeamento da classe Curso).

Mais detalhes

Sites:
Hibernate – http://www.hibernate.org/
Spring – http://www.springframework.org/
EhCahe - http://ehcache.sourceforge.net/

Referências:

Pro Spring. Harrop, Rob; Machacek, Jan. Editora Apress, 2005.
Hibernate In Action. Bauer, Christian; King, Gavin. Editora Manning, 2004.

Conclusão

Agora você já sabe como montar uma camada de persistência com o Hibernate, como integrar ele com o Spring e ainda como configurar o cache para aumentar ainda mais a performance da sua aplicação.

O Hibernate continua sendo a ferramenta de mapeamento objeto/relacional mais utilizada na comunidade Java, mas é sempre importante ficar de olhos abertos para as novidades do mercado. A especificação EJB 3.0 está praticamente pronta e segundo os seus defensores ela vai funcionar normalmente fora dos servidores de aplicação. Se a promessa se tornar realidade (o Hibernate já tem uma implementação da versão atual da implementação, o EntityManager) podemos esperar por uma verdadeira "guerra" entre os fornecedores de ferramentas para entrar no mercado e com essa concorrência nós só temos a ganhar.

Outro fator interessante é a configuração do Hibernate através de "annotations", que ainda está em fase de testes mas que em pouco tempo deve ser tornar uma maneira mais simples de se lidar com os quilos de configuração necessários para se utilizar em uma aplicação com o Hibernate (e aplicações Java em geral). Annotations ajudam mas continuam sendo apenas uma "mudança de lugar" para a configuração, que saiu dos arquivos XML e foi parar dentro do código Java, trazendo mais uma complicação, que é ter sempre que recompilar a aplicação quando mudanças ocorrerem, além de ser mais complicado adicionar "named queries" em arquivos de classes, pois elas realmente misturam código de acesso a banco de dados dentro das suas classes de modelo.

O exagero e o cansaço dos desenvolvedores com a edição de tantos "arquivos XML" é tão grande que novos frameworks simplesmente aboliram o uso de configuração e buscam informações do próprio código e organização da aplicação, é o movimento de "convenção sobre configuração" que está tomando "invadindo" os desenvolvedores cansados com tantos arquivos para manter. Vamos esperar que esse movimento realmente cresça na comunidade Java pra podermos diminuir a quantidade de configuração e nos preocuparmos mais com o que as aplicações devem realmente fazer.

Maurício Linhares de Aragão Junior (*mauricio.linhares@gmail.com*) é graduando em Desenvolvimento de Software para Internet no CEFET-PB e Comunicação Social (habilitação Jornalismo) na UFPB. É membro do Paraíba Java Users Group (PBJUG), do Grupo de Usuários GNU/Linux da Paraíba (GLUG-PB) e moderador dos fóruns do Grupo de Usuários Java (GUJ).