Documentação da Referência do Hibernate

Version: 3.2 ga

Índice

Prefácio viii
1. Introdução ao Hibernate 1
1.1. Prefácio 1
1.2. Parte 1 – A primeira aplicação Hibernate 1
1.2.1. A primeira Classe 1
1.2.2. O arquivo de mapeamento 3
1.2.3. Configuração do Hibernate 5
1.2.4. Compilando com o Ant 6
1.2.5. Startup e helpers 7
1.2.6. Carregando e salvando objetos 8
1.3. Parte 2 - Mapeando associações 11
1.3.1. Mapeando a classe Person 11
1.3.2. Uma associação unidirectional baseada em um Set 11
1.3.3. Trabalhando a associação 12
1.3.4. Coleção de valores 14
1.3.5. Associações bidirecionais 15
1.3.6. Trabalhando com associações bidirecionais 16
1.4. EventManager um aplicativo para internet 17
1.4.1. Criando um servlet básico 17
1.4.2. Processando e renderizando 18
1.4.3. Instalando e testando 19
1.5. Sumário 20
2. Arquitetura 21
2.1. Visão Geral 21
2.2. Estados de instância 23
2.3. Integração JMX 23
2.4. Suporte JCA 24
2.5. Sessões contextuais 24
3. Configuração 26
3.1. 1.11 Configuração programática 26
3.2. Obtendo uma SessionFactory 27
3.3. Conexões JDBC 27
3.4. Propriedades opcionais de configuração 28
3.4.1. Dialetos SQL 34
3.4.2. Recuperação por união externa (Outer Join Fetching) 35
3.4.3. Fluxos Binários (Binary Streams) 35
3.4.4. Cachê de segundo nível e consulta 35
3.4.5. Substituições na Linguagem de Consulta 35
3.4.6. Estatísticas do Hibernate 36
3.5. Logging 36
3.6. Implementando uma NamingStrategy 37
3.7. Arquivo de configuração XML 37
3.8. Integração com servidores de aplicação J2EE 38
3.8.1. Configuração de estratégia de transação 39
3.8.2. SessionFactory associada a JNDI 40
3.8.3. Administração de contexto de Sessão atual com JTA 40
3.8.4. Deployment JMX 40
4. Classes persistentes 42

4.1. Um exemplo simples de POJO 42
4.1.1. Implementar um constructor sem argumentos 43
4.1.2. Garanta a existencia de uma propriedade identificadora (opcional) 43
4.1.3. Prefira classes que não sejam marcadas como final(opcional) 44
4.1.4. Declare metodos accessores e modificadores para os campos persistentes (opcional) . 44
4.2. Implementando Herança 44
4.3. Implementando equals() e hashCode() 44
4.4. Modelos dinâmicos 45
4.5. Tuplas 47
5. Mapeamento O/R Bassico 49
5.1. Declaração de mapeamento 49
5.1.1. Doctype 50
5.1.1.1. EntityResolver 50
5.1.2. Mapeamento Hiberante 51
5.1.3. class 51
5.1.4. id 54
5.1.4.1. Generator 55
5.1.4.2. Algoritmo Hi/lo 56
5.1.4.3. UUID algorithm 56
5.1.4.4. Colunas de identidade e sequencias 56
5.1.4.5. Identificadores especificados 57
5.1.4.6. Primary keys geradas por triggers 57
5.1.5. composite-id 57
5.1.6. discriminator 58
5.1.7. version (opcional) 59
5.1.8. timestamp (opcional) 60
5.1.9. property 60
5.1.10. many-to-one 62
5.1.11. one-to-one (um-pra-um) 63
5.1.12. natural-id 65
5.1.13. componente, componente dinâmico. 65
5.1.14. propriedades 66
5.1.15. subclass (subclasse) 67
5.1.16. joined-subclass 68
5.1.17. union-subclass 69
5.1.18. join 69
5.1.19. key 70
5.1.20. elementos column e formula 71
5.1.21. import 71
5.1.22. any 72
5.2. Tipos do Hibernate 73
5.2.1. Entidades e valores 73
5.2.2. Valores de tipos básicos 73
5.2.3. Tipos de valores personalizados 74
5.3. Mapeando uma classe mais de uma vez 75
5.4. SQL quoted identifiers 76
5.5. Metadata alternativos 76
5.5.1. Usando marcação XDoclet 76
5.5.2. Usando anotações JDK 5.0 78
5.6. Propriedades geradas 79
5.7. Objetos auxiliares de banco de dados 79
6. Mapeamento de Coleções. 81

6.1. Persistent collections 81
6.2. Mapeamento de coleções 81
6.2.1. Collection foreign keys 83
6.2.2. Elementos da coleção 83
6.2.3. Coleções indexadas 83
6.2.4. Coleções de valores associações muitos-para-muitos 84
6.2.5. Associações um-para-muitos 86
6.3. Mapeamento avançado de coleções 87
6.3.1. Coleções ordenadas 87
6.3.2. Associações Bidirectionais 87
6.3.3. Associações bidirecionais com coleções indexadas 89
6.3.4. Associações Ternárias 90
6.3.5. Usando o <idbag> 90
6.4. Exemplos de coleções 91
7. Mapeamento de Associações. 94
7.1. Introdução 94
7.2. Associações Unidirecionais 94
7.2.1. muitos para um 94
7.2.2. um para um 94
7.2.3. um para muitos 95
7.3. Associações Unidirecionais com tabelas associativas 96
7.3.1. um para muitos 96
7.3.2. muitos para um 96
7.3.3. um para um 97
7.3.4. muitos para muitos 97
7.4. Associações Bidirecionais 98
7.4.1. um para muitos / muitos para um 98
7.4.2. um para um 99
7.5. Associações Bidirecionais com tabelas associativas 99
7.5.1. um para muitos / muitos para um 99
7.5.2. um para um 100
7.5.3. muitos para muitos 101
7.6. Mapeamento de associações mais complexas 101
8. Mapeamento de Componentes. 103
8.1. Objetos dependentes 103
8.2. Coleções de objetos dependentes 104
8.3. Componentes como índices de Map 105
8.4. . Componentes como identificadores compostos 106
8.5. Componentes Dinâmicos 107
9. Mapeamento de Herança 108
9.1. As três estratégias 108
9.1.1. Tabela por hierarquia de classes 108
9.1.2. Tabela por subclasse 109
9.1.3. Tabela por subclasse, usando um discriminador 109
9.1.4. Misturando tabela por hierarquia de classes com tabela por subclasse 110
9.1.5. Tabela por classe concreta 110
9.1.6. Tabela por classe concreta, usando polimorfismo implícito 111
9.1.7. Misturando polimorfismo implícito com outros mapeamentos de herança 112
9.2. Limitações 112
10. Trabalhando com objetos 114
10.1. Estado dos objetos no Hibernate 114
10.2. Tornando os objetos persistentes 114

10.3. Carregando o objetos 115
10.4. Consultando 116
10.4.1. Executando consultas 116
10.4.1.1. Interagindo com resultados 117
10.4.1.2. Consultas que retornam tuplas 117
10.4.1.3. Resultados escalares 117
10.4.1.4. Bind de parametros 118
10.4.1.5. Paginação 118
10.4.1.6. Paginação Interativa 118
10.4.1.7. Externalanzo consultas nomeadas 119
10.4.2. Filtrando coleções 119
10.4.3. Consultas por criterios 120
10.4.4. Consultas em sql nativo 120
10.5. Modificando objetos persistentes 120
10.6. Modificando objetos destacados 121
10.7. Detecção automática de estado 122
10.8. Deletando objetos persistentes 123
10.9. Replicando objtos entre base de da dados diferentes 123
10.10. Limpando a Session 124
10.11. Persistência transitiva 125
10.12. Usando metadados 126
11. Transações e Concorrência 127
11.1. Session e escopos de transações 127
11.1.1. Unidade de trabalho 127
11.1.2. Longas conversações 128
11.1.3. Considerando a identidade do objeto 129
11.1.4. Edições comuns 130
11.2. Demarcação de transações de bancos de dados 130
11.2.1. Ambiente não gerenciado 131
11.2.2. Usando JTA 132
11.2.3. Tratamento de Exceção 133
11.2.4. Timeout de Transação 134
11.3. Controle de concorrência otimista 134
11.3.1. Checagem de versão da aplicação 135
11.3.2. Sessão estendida e versionamento automático 135
11.3.3. Objetos destacados e versionamento automático 136
11.3.4. Versionamento automático customizado 136
11.4. Locking pessimista 137
11.5. Modos de liberar a Connection 138
12. Interceptadores e Eventos 139
12.1. Interceptadores 139
12.2. Sistema de Eventos 140
12.3. Segurança declarativa no Hibernate 141
13. Processamento de lotes 143
13.1. Inserção de lotes 143
13.2. Batch updates 143
13.3. A interface StatelessSession 144
13.4. Operações no estilo DML 144
14. HQL: A linguagem de Queries do Hibernate 147
14.1. Case Sensitíve 147
14.2. A clausula from 147
14.3. Associações e joins 147

14.4. Formas e sintaxe de joins 149
14.5. Clausula select 149
14.6. Funções de agregação 150
14.7. Queries polimórficas 150
14.8. A clausula where 151
14.9. Expressões 152
14.10. A clausula order by 155
14.11. A clausula group by 155
14.12. Subqueries 156
14.13. Exemplos de HQL 157
14.14. update e delete em lote 158
14.15. Dicas e Truques 158
15. Consultas por critérios 160
15.1. Criando uma instancia Criteria 160
15.2. Limitando o result set 160
15.3. Ordenando os resultados 161
15.4. Associações 161
15.5. Recuperação dinamica de associações 162
15.6. Consultas por exemplo 162
15.7. Projections, aggregation and grouping 163
15.8. Consultas e sub consultas separadas 164
15.9. Consultas por identificador natural 164
16. SQL nativo 166
16.1. Usando o SQLQuery 166
16.1.1. Consultas escalres 166
16.1.2. Entity queries 167
16.1.3. Manipulando associações e coleções 167
16.1.4. Retornando múltiplas Entidades 168
16.1.4.1. Alias and property references Aliases e referências de propriedade 168
16.1.5. Retornando Entidades não gerenciads 169
16.1.6. Controlando a herança 169
16.1.7. Parâmetros 170
16.2. Consultas SQL com nomes 170
16.2.1. Usando a propriedade retornada para especificar explicitamente os nomes de colunas e aliás 171
16.2.2. Usando stored procedures para consultas 172
16.2.2.1. Regras/limitações no uso de stored procedures 172
16.3. SQL customizado para create, update e delete 173
16.4. SQL customizado para carga 174
17. Filtrando dados 176
17.1. Filtros do Hibernate 176
18. Mapeamento XML 178
18.1. Trabalhando com dados em XML 178
18.1.1. Especificando o mapeamento de uma classe e de um arquivo XML simultaneamente 178
18.1.2. Especificando somente um mapeamento XML 178
18.2. Mapeando metadados com XML 179
18.3. Manipulando dados em XML 180
19. Aumentando a performance 182
19.1. Estratégias de Fetching 182
19.1.1. Trabalhando com associações preguiçosas (lazy) 183
19.1.2. Personalizando as estratégias de recuperação 183
19.1.3. Proxies de associação single-ended 184

19.1.4. Inicializando coleções e proxies 185
19.1.5. Usando busca em lote 186
19.1.6. Usando subselect fetching 187
19.1.7. Usando busca preguiçosa de propriedade 187
19.2. O Cache de segundo nível 188
19.2.1. Mapeamento do cache 189
19.2.2. Strategia: somente leitura 189
19.2.3. Strategia: read/write 189
19.2.4. Estratégia: nonstrict read/write 190
19.2.5. Estratégia: transactional 190
19.3. Administrando os caches 190
19.4. O caché de consultas 191
19.5. Entendendo a performance de coleções 192
19.5.1. Taxonomania 192
19.5.2. Lists, maps, idbags e sets são as coleções mais eficientes de se atualizar 193
19.5.3. Bags e lists são as coleções inversas mais eficientes 193
19.5.4. Deletando tudo de uma vez 194
19.6. Monitorando o desempenho 194
19.6.1. Monitorando a SessionFactory 194
19.6.2. Métricas 195
20. Guia de ferramentas 197
20.1. Geração automática de schema 197
20.1.1. Personalizando o schema 197
20.1.2. Rodando a ferramenta 199
20.1.3. Propriedades 200
20.1.4. Usando o Ant 200
20.1.5. Ataulizações Incrementais do schema 201
20.1.6. Usando o Ant para updates incrementais do schema 201
20.1.7. Validação do Schema 202
20.1.8. Usando o Ant para a validação de schema 202
21. Exemplo: Mestre/Detalhe 203
21.1. Uma nota sobre coleções 203
21.2. Um-para-muitos bidirectional 203
21.3. ciclo de vida em cascata 205
21.4. Tratamento em Cascata e unsaved-value 205
21.5. Conclusão 206
22. Exemplo: Aplicação Weblog 207
22.1. Classes persistentes 207
22.2. Mapeamentos Hibernate 208
22.3. Código Hibernate 209
23. Exemplo: Vários Mapeamentos 213
23.1. Employer/Employee 213
23.2. Author/Work 214
23.3. Customer/Order/Product 216
23.4. Exemplos variados de mapeamento 218
23.4.1. Associação um-para-um "Tipadas" 218
23.4.2. Exemplo de chave composta 218
23.4.3. Muitos-para-muitos com atributo de chave composta compartilhada 220
23.4.4. Conteúdo baseado em descriminação 221
23.4.5. Associações em chaves alternativas 221
24. Boas práticas 223

Prefácio

Advertência! Esta é uma versão traduzida do inglês da documentação de referência do Hibernate. A versão traduzida pode estar desatualizada. Caso existam, as diferenças devem ser pequenas e serão corrigidas o mais breve possível. Caso esteja faltando alguma informação ou você encontre erros, consulte a documentação de referência em inglês e, se quiser colaborar com a tradução, entre em contato com um dos tradutores abaixo:. Gamarra

Tradutor(es) em ordem alfabética:

- *Alvaro Netto* - alvaronetto@cetip.com.br
- *Anderson Braulio* - andersonbraulio@gmail.com
- *Daniel Vieira Costa* - danielvc@gmail.com
- *Anne Carolinne* - carolinnecarvalho@gmail.com
- *Francisco gamarra* - francisco.gamarra@gmail.com
- *Gamarra* - mauricio.gamarra@gmail.com
- *Luiz Carlos Rodrigues* - luizcarlos_rodrigues@yahoo.com.br
- *Marcel Castelo* - marcel.castelo@gmail.com
- *Marvin Herman Froeder* - m@rvin.info
- *Pablo L. de Miranda* - pablolmiranda@gmail.com
- *Paulo César* - paulocol@gmail.com
- *Renato Deggau* - rdeggau@gmail.com
- *Rogério Araújo* - rgildoaraujo@yahoo.com.br
- *Wanderson Siqueira* - wandersonxs@gmail.com

Quando trabalhamos com software orientado a objetos e banco de dados relacional, podemos ter alguns incômodos em ambientes empresariais. O Hibernate é uma ferramenta que faz o mapeamnto objeto/relacional no ambiente Java. O termo de mapeamento de objeto/relacional (ou ORM # Object/Relational Mapping) se refere a técnica de mapear uma representação de dados de um modelo de objeto para dados de modelo relacional com o esquema baseado em SQL

O Hibernate não somente cuida do mapeamento de classes Java para tabelas de banco de dados (e de tipos de dados em Java para tipos de dados em SQL), como também fornece facilidade de consultas e recuperação de dados, podendo também reduzir significantemente o tempo de desenvolvimento gasto com a manipulação manual de dados no SQL e JDBC.

O objetivo do Hibernate é aliviar o desenvolvedor de 95 por cento das tarefas comuns de programação relacionadas a persistência de dados. O Hibernate talvez não seja a melhor solução para aplicações que usam somente stored procedures para implementar a lógica de negócio no banco de dados, isto é muito utilizado com o domínio de modelos orientado a objetos e lógicas de negócio em camadas do meio (middle-tier) baseadas em Java. Porém, o Hibernate certamente poderá ajudá-lo a remover ou encapsular o código SQL de um banco de dados

específico, ajudando também com a tarefa comum da tranformação de um result set para um gráfico de objetos.

Se você for novo no Hibernate e no mapeamento Objeto/Relacional, ou até mesmo em Java, por favor, siga os seguintes passos:

1. Leia o Capítulo 1, *Introdução ao Hibernate* para um tutorial com instruções passo-a-passo. O código fonte para do tutorial está incluído na distribuição no diretório `doc/reference/tutorial/`.

2. Leia o Capítulo 2, *Arquitetura* para entender o ambiente onde o Hibernate pode ser utilizado.

3. Dê uma olhada no diretório de exemplo `eg/` da distribuição do Hibernate, ele contém uma aplicação standalone simples. Copie seu driver JDBC para o diretório `lib/` e edite o arquivo `etc/hibernate.properties`, especificando corretamente os valores para seu banco de dados. Usando o prompt de comando no diretório de distribuição, digite `ant eg` (usando Ant), ou no Windows, digite `build eg`.

4. Use esta documentação de referência como sua fonte primária de informação. Considere ler também o livro *Hibernate in Action* (http://www.manning.com/bauer) caso você precise de mais ajuda com o desenvolvimento de aplicações ou caso prefira um tutorial passo-a-passo. Também visite o site http://caveatemptor.hibernate.org e faça o download da aplicação de exemplo do Hibernate em Ação.

5. FAQs (perguntas feitas com mais freqüência) estão respondidas no site do Hibernate

6. Demonstrações, exemplos e tutoriais estão disponíveis no site do Hibernate.

7. A Área da comunidade no site do Hibernate é uma boa fonte de recursos para padrões de projeto e várias soluções de integração (Tomcat, JBoss AS, Struts, EJB, etc.).

Caso você tenha dúvidas, use o fórum dos usuários encontrado no site do Hibernate. Nós também fornecemos um sistema para controle de bugs (JIRA) para relatórios de erros e requisições de features. Se você está interessado no desenvolvimento do Hibernate, junte-se a lista de e-mail dos desenvolvedores.

Suporte comercial de desenvolvimento, suporte de produção e treinamento para o Hibernate está disponível através do JBoss Inc. (veja http://www.hibernate.org/SupportTraining). O Hibernate é um Projeto Profissional de Código Aberto e um componente crítico da suíte de produtos JBoss Enterprise Middleware System (JEMS).

Capítulo 1. Introdução ao Hibernate

1.1. Prefácio

Este capítulo é um tutorial introdutório para novos usuários do Hibernate. Nós iniciaremos com uma simples aplicação de linha de comando usando uma base de dados em memória tornando um passo de fácil de compreender.

Este tutorial é voltado para novos usuários do Hibernate, mas requer um conhecimento de Java e SQL. Este tutorial é baseado no tutorial de Michael Gloegl, as bibliotecas Third Party foram nomeadas para JDK 1.4 e 5.0. Você pode precisar de outras bibliotecas para JDK 1.3.

O código fonte do tutorial está incluído no diretório da distribuição `doc/reference/tutorial/`.

1.2. Parte 1 – A primeira aplicação Hibernate

Primeiro, nós iremos criar uma simples aplicação Hibernate baseada em console. Usaremos uma base de dados Java (HSQL DB), então não teremos que instalar nenhum servidor de banco de dados.

Vamos supor que precisemos de uma aplicação com um banco de dados pequeno que possa armazenar e atender os eventos que queremos, e as informações sobre os hosts destes eventos.

A primeira coisa que devemos fazer é configurar nosso diretório de desenvolvimento, e colocar todas as bibliotecas Java que precisamos dentro dele. Faça o download da distribuição do Hibernate no site. Descompacte o pacote e coloque todas as bibliotecas necessárias encontradas no diretório `/lib`, dentro do diretório `/lib` do seu novo projeto. Você deverá ter algo parecido com isso:

```
.
+lib
  antlr.jar
  cglib.jar
  asm.jar
  asm-attrs.jars
  commons-collections.jar
  commons-logging.jar
  hibernate3.jar
  jta.jar
  dom4j.jar
  log4j.jar
```

Esta é a configuração mínima requerida das bibliotecas (observe que também foi copiado o hibernate3.jar da pasta principal do Hibernate) para o Hibernate *na hora do desenvolvimento*. O Hibernate permite que você utilize mais ou menos bibliotecas. Veja o arquivo `README.txt` no diretório `lib/` da distribuição do Hibernate para maiores informações sobre bibliotecas requeridas e opcionais. (Atualmente, a biblioteca Log4j não é requerida, mas é a preferida de muitos desenvolvedores.)

Agora, iremos criar uma classe que representa o evento que queremos armazenar na base de dados..

1.2.1. A primeira Classe

Nossa primeira classe de persistência é uma simples classe JavaBean com algumas propriedades:

```
package events;
```

```
import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

Você pode ver que esta classe usa o padrão JavaBean convencional de nomes para os métodos getters e setters das propriedades, como também a visibilidade private dos campos. Este é um padrão de projeto recomendado, mas não obrigatório. O Hibernate pode também acessar campos diretamente, o benefício para os métodos de acesso é a robustez para o Refactoring. O construtor sem argumento é obrigatório para instanciar um objeto desta classe por meio de reflexão.

A propriedade `id` mantém um valor único de identificação para um evento em particular. Todas as classes persistentes de entidade (bem como aquelas classes dependentes de menos importância) precisam de uma propriedade de identificação, caso nós queiramos usar o conjunto completo de características do Hibernate. De fato, a maioria das aplicações (em especial aplicações web) precisam destinguir os objetos pelo identificador, então você deverá considerar esta, uma característica em lugar de uma limitação. Porém, nós normalmente não manipulamos a identidade de um objeto, consequentemente o método setter deverá ser privado. O Hibernate somente atribuirá valores aos identificadores quando um objeto for salvo. Você pode ver como o Hibernate pode acessar métodos públicos, privados, e protegidos, como também campos (públicos, privados, protegidos) diretamente. A escolha será sua, e você pode ajustá-la a sua aplicação.

O construtor sem argumentos é um item obrigatório para todas as classes persistentes; O Hibernate tem que criar para você os objetos usando a reflection do Java. O construtor pode ser privado, porém, a visibilidade miníma de pacote é obrigatóira para a geração de proxies em tempo de execução e recuperação eficiente dos dados sem a instrumentação de bytecode.

Coloque este fonte Java no diretório chamado `src` na pasta de desenvolvimento, e em seu pacote correto. O diretório deverá ser parecido como este:

```
.
+lib
  <Hibernate and third-party libraries>
```

```
+src
  +events
    Event.java
```

No próximo passo, iremos falar sobre as classes de persistência do Hibernate.

1.2.2. O arquivo de mapeamento

O Hibernate precisa saber como carregar e armazenar objetos da classe de persistência. Este é o ponto onde o arquivo de mapeamento do Hibernate entra em cena. O arquivo de mapeamento informa ao Hibernate, qual tabela no banco de dados ele deverá acessar, e quais as colunas na tabela ele deverá usar.

A estrutura básica de um arquivo de mapeamento é parecida com:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
[...]
</hibernate-mapping>
```

Veja que o DTD do Hibernate é muito sofisticado. Você pode usa-lo no mapeamento XML para auto-completar os elementos e atributos no seu editor ou IDE. Você também pode abrir o arquivo DTD no seu editor – é a maneira mais fácil de ter uma visão geral de todos os elementos e atributos e dos valores padrões, como também alguns comentários. Veja que o Hibernate não irá carregar o arquivo DTD da web, e sim do diretório da aplicação (classpath). O arquivo DTD está incluído no `hibernate3.jar` como também no diretório `src/` da distribuição do Hibernate.

Nós omitiremos a declaração do DTD nos próximos exemplos para encurtar o código. Isto, é claro, não é opcional.

Entre as duas tags `hibernate-mapping`, inclua um elemento `class`. Todas as classes persistentes da entidade (novamente, poderá haver mais tarde, dependências entre as classes que não são classes-primárias de entidades) necessitam do tal mapeamento, para uma tabela no banco de dados SQL.

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">

    </class>

</hibernate-mapping>
```

Mais adiante iremos dizer ao Hibernate como fazer para persistir e carregar objetos da classe `Event` da tabela `EVENTS`, cada instancia representada por uma lina na tabela. Agora, continuaremos com o mapeamento de uma única propriedade identificadora para as primary keys da tabela. Além disso, nós não iremos nos importar com esta propriedade identificadora, nós iremos configurar uma estratégia de geração de id's para uma primary key de uma surrogate key:

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>
```

```
</hibernate-mapping>
```

O elemento `id` é a declaração da propriedade identificadora, o `name="id"` declara o nome da propriedade Java – o Hibernate irá usar os métodos getter e setter para acessar a propriedade. O atributo da coluna informa ao Hibernate qual coluna da tabela `EVENTS` nós iremos usar como primary key. O elemento `generator` especifica a estratégia de geração do identificador, neste caso usaremos `native`, que escolhe a melhor estratégia dependendo do banco de dados (dialeto) configurado. O Hibernate suporta identificadores gerados pelo banco de dados, identificaores únicos, é e claro, identificadores atribuidos na aplicação (ou qualquer outra estratégia através de uma extensão).

Finalmente incluiremos as declarações para as propriedades persistentes da classe no arquivo de mapeamento. Por default, nenhuma das propriedades da classe é considerada persistente:

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

Da mesma maneira que com o elemento `id`, o atributo `name` do elemento `property` informa ao Hibernate qual método getter e setter deverá usar. Assim, neste caso, o Hibernate irá procurar pelo `getDate()/setDate()`, como também pelo `getTitle()/setTitle()`.

Porque fazer o mapeamento da propriedade `date` incluído no atributo `column`, e na propriedade title não? Sem o atributo `column` o Hibernate por padrão usa o nome da propriedade como o nome da coluna. Isso funciona muito bem para a propriedade `title`. Entretanto a propriedade `date` é uma palavra-chave reservada na maioria dos bancos de dados, assim nós detalhamos o mapeamento indicando um nome de coluna.

Outra coisa interessante é que no mapemanto de `title` também falta o atributo `type`. O tipo que declaramos e usamos nos arquivos de mapeamento, não são como você pode esperar, tipos de dados Java. Eles tambem não são tipos de dados SQL. Esses tipos podem ser chamados de *Tipos de mapeamento Hibernate*, que são conversores que podem traduzir tipos de dados do Java para os tipos de dados SQL e vice-versa. Novamente, o Hibernate tentará determinar a conversão correta e mapeará o o tipo apropriado, caso o atributo `type` não esteja presente no mapeamento. Em alguns casos, esta detecção automática (que usa Reflection sobre as classes Java) pode não ter o resultado que você espera ou necessita. Este é o caso com a propriedade `date`. O Hibernate não pode saber se a propriedade (que é do `java.util.Date`) deve ser mapeada para uma coluna SQL do tipo `date`, `timestamp`, ou `time`. Nós salvamos as informaçõs completas de datas e horas mapeando a propriedade com um `timestamp`.

Este arquivo de mapeamento deve ser salvo com o nome `Event.hbm.xml`, no mesmo diretório que o arquivo fonte da Classe Java `Event`. A escolha do nome dos arquivos de mapeamento pode ser arbitrário, porém o sufixo `hbm.xml` é uma convenção da comunidade dos desenvolvedores do Hibernate. Sua estrutura do diretório deve agora se parecer com isso:

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
    Event.hbm.xml
```

Nós iremos continuar com a configuração principal do Hibernate.

1.2.3. Configuração do Hibernate

Agora nós temos uma classe persistente e seu arquivo de mapeamento no lugar. Está na hora de configurar o Hibernate. Antes de fazermos isso, iremos precisar de um banco de dados. O HSQL DB, um SQL DBMS feito em java, pode ser baixado através do site do HSQL DB. Atualmente, você só precisa baixar o `hsqldb.jar`. Coloque este arquivo no diretório da pasta de desenvolvimento `lib/`.

Crie um diretório chamado `data` no diretório root de desenvolvimento – Este será onde o HSQL DB irá armazenar arquivos de dados. Agora iremos iniciar o banco de dados executando `java -classpath ../lib/hsqldb.jar org.hsqldb.Server` neste diretório de dados. Você pode ver ele iniciando e conectando ao socket TCP/IP, será onde nossa aplicação irá se conectar. Se você deseja iniciar uma nova base de dados durante este tutorial, finalize o HSQL DB(pressionando o `CTRL + C` na janela), delete todos os arquivos no diretório `data/`, e inicie o HSQL BD novamente.

O Hibernate é a camada da sua aplicação que se conecta com o banco de dados, para isso necessita de informação da conexão. As conexões são feitas através de um pool de conexões JDBC, a qual teremos que configurar. A distribuição do Hibernate contém diversas ferramentas de pooling da conexão JDBC open source, mas iremos usar o pool de conexão interna para este tutorial. Note que você tem que copiar as bibliotecas necessárias para seu classpath e usar configurações diferentes para o pooling de conexão caso você deseje utilizar um software de pooling JDBC de terceiros com qualidade de produção.

Para as configurações do Hibernate, nós podemos usar um arquivo simples `hibernate.properties`, um arquivo ligeiramente mais sofisticado `hibernate.cfg.xml` ou até mesmo uma instalação programática completa. A maioria dos usuários preferem utilizar o arquivo de configuração XML

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <mapping resource="events/Event.hbm.xml"/>
```

```
    </session-factory>

</hibernate-configuration>
```

Veja que esta configuração XML usa um DTD diferente. Nós configuraremos as `SessionFactory` do Hibernate – uma factory global responsável por uma base de dedados particular. Se você tiver diversas bases de dados, use diversas configurações `<session-factory>`, geralmente em diversos arquivos de configuração (para um inicio mais fácil).

As primeiras quatro `propriedades` do elemento contêm a configuração necessária para a conexão ao JDBC. A `propriedade` dialect do elemento especifica a variante particular do SQL que o Hibernate gera. O gerenciamento automático de sessão do Hibernate para contextos de persistência estará disponível em breve. A opção `hbm2ddl.auto` habilita a geração automática de schemas de banco de dados – diretamente na base de dados. Isso naturalmente também pode ser desligado (removendo a opção da configuração) ou redirecionando para um arquivo com ajuda do `SchemaExport` nas tasks do Ant. Finalmente, iremos adicionar os arquivos das classes de persistência mapeadas na configuração.

Copie este arquivo no diretório fonte, assim isto irá terminar na raiz (root) do classpath. O Hibernate automaticamente procura por um arquivo chamado `hibernate.cfg.xml` na raiz do classpath, no startup.

1.2.4. Compilando com o Ant

Nos iremos, agora, compilar o tutorial com Ant. Você ira precisar do Ant instalado – se encontra disponível na página de download do Ant [http://ant.apache.org/bindownload.cgi]. Como instalar o Ant, não será abordado aqui. Caso tenha alguma dúvida, por favor, acesso o manual do Ant [http://ant.apache.org/manual/index.html]. Depois que tiver instalado o Ant, podemos começar a criar o arquivo `build.xml`. Este arquivo será chamado de `build.xml` e colocado no diretório de desenvolvimento.

Um arquivo básico de build, se parece com isto:

```
<project name="hibernate-tutorial" default="compile">

    <property name="sourcedir" value="${basedir}/src"/>
    <property name="targetdir" value="${basedir}/bin"/>
    <property name="librarydir" value="${basedir}/lib"/>

    <path id="libraries">
        <fileset dir="${librarydir}">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${targetdir}"/>
        <mkdir dir="${targetdir}"/>
    </target>

    <target name="compile" depends="clean, copy-resources">
      <javac srcdir="${sourcedir}"
             destdir="${targetdir}"
             classpathref="libraries"/>
    </target>

    <target name="copy-resources">
        <copy todir="${targetdir}">
            <fileset dir="${sourcedir}">
                <exclude name="**/*.java"/>
            </fileset>
        </copy>
    </target>
```

```
</project>
```

Isto irá dizer ao Ant para adicionar todos os arquivos no diretório lib terminando com `.jar`, no classpath usado para compilação. Irá também irap copiar todos os arquivos não-java para o diretório de compilação (arquivos de configuração, mapeamento). Se você executar o ant agora, deverá ter esta saída.

```
C:\hibernateTutorial\>ant
Buildfile: build.xml

copy-resources:
     [copy] Copying 2 files to C:\hibernateTutorial\bin

compile:
    [javac] Compiling 1 source file to C:\hibernateTutorial\bin

BUILD SUCCESSFUL
Total time: 1 second
```

1.2.5. Startup e helpers

É hora de recuperar e salvar alguns objetos `Event`, mas primeiro nós temos de completar o setup com algum código de infraestrutura. Este startup inclui a construção de um objeto `SessionFactory` global e coloca-lo em algum lugar de fácil acesso para o código da aplicação. Uma `SessionFactory` pode abrir novas `Session`'s. Uma `Session` representa uma unidade de trabalho de theaded simples , a `SessionFactory` é um objeto global thread-safe, instanciado uma vez.

Nos iremos criar uma classe helper `HibernateUtil`, que toma conta do startup e faz acesso a uma `SessionFactory` de maneira conveniente. Vamos dar uma olhada na implementação:

```
package util;

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

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

Esta classe não só produz a `SessionFactory` global no seu código de inicialização estático. (chamado uma vez pela JVM quando a classe é carregada), mas também esconde o fato de que isto usa um static singleton. Ela pode muito bem, enxergar a `SessionFactory` do JNDI em um application server.

Se você der à `SessionFactory` um nome, no seu arquivo de configuração. O Hibernate irá, de fato, tentar associa-lo ao JNDI depois que estiver construído. Para evitar completamente este código, você também poderia usar

o deployment do JMX e deixar o contêiner JMX, instanciar e associar a um `HibernateService` no JNDI. Essas opções avançadas são discutidas no documento de referência do Hibernate.

Coloque o `HibernateUtil.java` no diretório de arquivos de desenvolvimento(source), em um pacote após o `events`:

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
    Event.hbm.xml
  +util
    HibernateUtil.java
  hibernate.cfg.xml
+data
build.xml
```

Novamente, isto deve compilar sem problemas. Finalmente, nós precisamos configurar um sistema de logging – o Hibernate usa commons logging e deixa você escolher entre o Log4j e o logging do JDK 1.4 . A maioria dos desenvolvedores prefere o Log4j: copie `log4j.properties` da distribuição do Hibernate (está no diretório `etc/`), para seu diretório `src`, depois vá em hibernate.cfg.xml. Dê uma olhada no exemplo de configuração e mude as configurações se você quizer ter uma saída mais detalhada. Por default, apenas as mensagems de start-tup e shwwn do Hibernate é mostrada no stdout.

O tutorial de infra-estrutura está completo - e nós já estamos preparados para algum trabalho de verdade com o Hibernate.

1.2.6. Carregando e salvando objetos

Finalmente, nós podemos usar o Hibernate para carregar e salvar algums objetos. Nós escrevemos uma classe `EventManager` com um método main():

```
package events;
import org.hibernate.Session;

import java.util.Date;

import util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {

        Session session = HibernateUtil.getSessionFactory().getCurrentSession();

        session.beginTransaction();

        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);

        session.save(theEvent);
```

```
        session.getTransaction().commit();
    }

}
```

Nós criamos um novo objeto `Event`, e passamos para o Hibernate. O Hibernate sabe como gerar o SQL e executar `INSERT`s no banco de dados. Vamos dar uma olhada na `Session` e no código de tratamento de `Transação` antes de executarmos o aplicativo.

Uma `Session` é uma unidade simples de trabalho. Por agora nós iremos começar com coisas simples e assumir uma granularidade de um-pra-um entre uma `Session` do Hibernate e uma transação de banco de dados. Para proteger nosso código do sistema de transação atual (nesse caso JDBC puro, mas também poderia rodar com JTA), nos usamos a API `Transaction`, que está disponível na `Session` do Hibernate.

O que a `sessionFactory.getCurrentSession()` faz? Primeiro, você pode chamar quantas vezes e de onde quiser, uma vez você tem sua `SessionFactory` (fácilmente, graças ao `HibernateUtil`). O método `getCurrentSession()` sempre retorna a unidade de trabalho "corrente". Você se lembra que nós mudamos a opção de configuração desse mecanismo para "thread" no `hibernate.cfg.xml`? Daqui em diante, o escopo da unidade de trabalho é a thread corrente do Java onde nossa aplicação é executada. Entretanto, esta não é toda a verdade, você também tem que considerar o escopo, quando uma unidade do trabalho começa e quando termina.

Uma `Session` é iniciada quando é usada pela primeira vez, quando é feita a primeira chamada à `getCurrentSession()`. Ela é então limitada pelo Hibernate à thread corrente. Quando a transação termina, tanto com commit quanto com rollback, o Hibernate também desassocia a `Session` da thread e a fecha pra você. Se você chamar `getCurrentSession()` novamente, você receberá uma nova `Session` e poderá iniciar uma nova unidade de trabalho. Esse modelo de programação *associado e limitidado thread*, é o modo mais popular de se usar o Hibernate, e permite uma modularização bastante flexivel do seu código (a demarcação de transação fica separada do código de acesso à dados, nós faremos isso mais adiante nesse tutorial).

Após relacionada à unidade ao escopo do trabalho, a `Session` o Hibernate deve ser usada para executar uma ou diversas operações na base de dados? O exemplo acima usa uma `Session` para uma operação. Isso foi coincidência, o exemplo não é complexo o bastante para se mostrar uma outra abordagem. O escopo de uma `Session` do Hibernate é flexível mas você não deve nunca projetar sua aplicação para usar uma nova `Session` do Hibernate para *cada* operação na base de dados. Assim mesmo que você veja isso nos exemplos seguintes (muito básicos), considere *session-per-operation* um anti-pattern. Uma aplicação (web) real é mostrada mais tarde neste tutorial.

Dê uma olhada no Capítulo 11, *Transações e Concorrência* para mais informações a respeito de manipulação e demarcação de transação. Nós também omitimos qualquer tratamento de erro e rollback no exemplo anterior.

Para executar esta rotina, nos teremos que adicionar uma chamada no arquivo build do Ant:

```
<target name="run" depends="compile">
    <java fork="true" classname="events.EventManager" classpathref="libraries">
        <classpath path="${targetdir}"/>
        <arg value="${action}"/>
    </java>
</target>
```

O valor do argumento `action`, é setado na linha de comando quando chamando esse ponto:

```
C:\hibernateTutorial\>ant run -Daction=store
```

Você deverá ver, após a compilação, o startup do Hibernate e, dependendo da sua configuração, muito log de saída. No final você verá a seguinte linha:

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)
```

Este é o `INSERT` executado pelo Hibernate, os pontos de interrogação representam parêmetros no estilo JDBC. Para ver os valores dos argumentos, ou para diminuir a verbosidade do log, veja o arquivo `log4j.properties`.

Agora nós gostaríamos de listar os eventos arquivados, então nós adicionamos uma opção para o método main:

```
if (args[0].equals("store")) {
    mgr.createAndStoreEvent("My Event", new Date());
}
else if (args[0].equals("list")) {
    List events = mgr.listEvents();
    for (int i = 0; i < events.size(); i++) {
        Event theEvent = (Event) events.get(i);
        System.out.println("Event: " + theEvent.getTitle() +
                           " Time: " + theEvent.getDate());
    }
}
```

Nos também adicionamos um novo `método listEvents()`:

```
private List listEvents() {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();

    session.beginTransaction();

    List result = session.createQuery("from Event").list();

    session.getTransaction().commit();

    return result;
}
```

O que nós fazemos aqui, é usar uma consulta HQL (Hibernate Query Language), para carregar todos os objetos `Events` exitentes no banco de dados. O Hibernate irá gerar o SQL apropriado, enviar para o banco de dados e popular objetos `Event` com os dados. Você pode criar queries mais complexas com HQL, claro.

Agora, para executar e testar tudo isso, siga os passos a seguir:

- Execute `ant run -Daction=store` para armazenar algo no banco de dados e, claro, gerar o esquema do banco de dados antes pelo hbm2ddl.
- Agora desabilite hbm2ddl comentando a propriedade no arquivo `hibernate.cfg.xml`. Normalmente só se deixa habilitado em teste unitários contínuos, mas outra carga de hbm2ddl pode *remover* tudo que você já tenha sido salvo. Sua configuração `create`, atualmente é traduzidas para "apague todas as tabelas do esquema, então recrie todas quando a SessionFactory estiver pronta".

Se você agora chamar o Ant com `-Daction=list`, você deverá ver os eventos que você acabou de criar. Você pode também chamar a ação `store` mais algumas vezes.

Nota: A maioria dos novos usuários do Hibernate falha nesse ponto e nós regularmente, vemos perguntas sobre mensagens de erro de *tabela não encontrada* . Entretanto, se você seguir os passos citados acima, você não terá esse problema, com o hbm2ddl criando o esquema do banco de dados na primeira execução, e restarts subsequentes da aplicação irão usar este esquema. Se você mudar o mapeamento e/ou o esquema do banco de dados, terá que habilitar o hbm2ddl novamente.

1.3. Parte 2 - Mapeando associações

Nós mapeamos uma classe de entidade de persistência para uma tabela. Agora vamos continuar e adicionar algumas associações de classe. Primeiro nos iremos adicionar pessoas a nossa aplicação, e armazenar os eventos de que elas participam.

1.3.1. Mapeando a classe Person

O primeiro código da classe `Person` é simples:

```
package events;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id'

}
```

Crie um novo arquivo de mapeamento, chamado `Person.hbm.xml` (não esqueça a referencia ao DTD no topo)

```
<hibernate-mapping>

    <class name="events.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
```

Finalmente, adicione o novo mapeamento a configuração do Hibernate:

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

Nos agora criaremos uma associação entre estas duas entidades. Obviamente, pessoas (Person) podem participar de eventos, e eventos possuem participantes. As questões de design com que teremos de lidar são: direcionalidade, multiplicidade e comportamento de coleção.

1.3.2. Uma associação unidirectional baseada em um `Set`

Nos iremos adicionar uma coleção de eventos na classe `Person`. Desse jeito poderemos navegar pelos eventos de uma pessoa em particular, sem executar uma query explicitamente – apenas chamando `aPerson.getEvents()`. Nos usaremos uma coleção Java, um `Set`, porquê a coleção não conterá elementos duplicados e a ordem não é relevante para nós.

Vamos escrever o código para isto nas classes Java e então fazer o mapeamento:

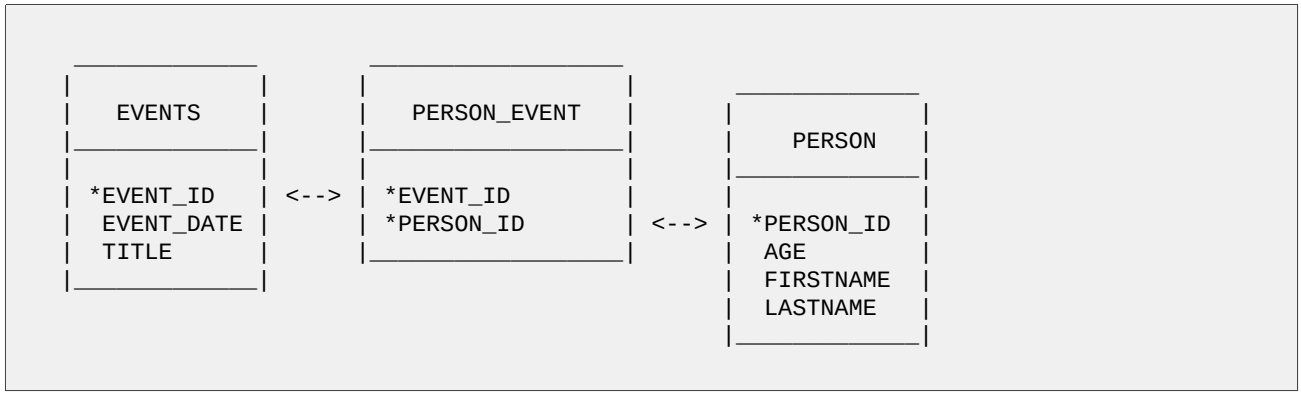
```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

    public void setEvents(Set events) {
        this.events = events;
    }
}
```

Antes de mapearmos esta associação, pense no outro lado. Claramente, poderíamos apenas fazer isto de forma unidirecional. Ou poderíamos criar outra coleção no `Event`, se quisermos ser capaz de navegar bidirecionalmente, i.e. um - `anEvent.getParticipants()`. Isto não é necessário, da perspectiva funcional. Você poderia sempre executar uma query explicita que retornasse os participantes de um evento em particular. Esta é uma escolha de design que cabe a você, mas o que é claro nessa discussão é a multiplicidade da associação: "muitos" valores em ambos os lados, nós chamamos isto uma associação *muitos-para-muitos*. Daqui pra frente, nos usaremos o mapeamento muitos-para-muitos do Hibernate:

```
<class name="events.Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>

    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="events.Event"/>
    </set>

</class>
```

O Hibernate suporta todo tipo de mapeamento de coleção, sendo um `<set>` mais comum. Para uma associação muitos-para-muitos (ou relacionamento de entidade *n:m*), uma tabela associativa é necessária. Cada linha nessa tabela representa um link entre uma pessoa e um evento. O nome da tabela é configurado com o atributo `table` do elemento `set`. O nome da coluna identificadora na associção, peloo lado da pessoa, é definido com o elemento `<key>`, o nome da coluna pelo lado dos eventos, e definido com o atributo `column` do `<many-to-many>`. Você também precisa dizer para o Hibernate a classe dos objetos na sua coleção (a classe do outro lado das coleções de referência).

Eis o esquema de mapeamento para o banco de dados:



1.3.3. Trabalhando a associação

Vamos recupear algumas pessoas e eventos ao mesmo tempo em um novo método na classe `EventManager`:

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    Event anEvent = (Event) session.load(Event.class, eventId);

    aPerson.getEvents().add(anEvent);

    session.getTransaction().commit();
}
```

Após carregar um `Person` e um `Event`, simplesmente modifique a coleção usando os métodos normais de uma coleção. Como você pode ver, não há chamada explícita para `update()` ou `save()`, o Hibernate detecta automaticamente que a coleção foi modificada e precisa ser atualizada. Isso é chamado de *checagem automática de sujeira*, e você também pode usá-la modificando o nome ou a data de qualquer um dos seus objetos. Assim quando eles estiverem no estado *persistent*, ou seja, limitado por uma `Session` do Hibernate em particular (i.e. eles foram carregados ou salvos dentro de uma unidade de trabalho), o Hibernate monitora qualquer alteração e executa o SQL em modo de escrita em segundo plano. O processo de sincronização do estado da memória com o banco de dados, ocorre geralmente apenas no final de uma unidade de trabalho, é chamado de *flushing*. No nosso código, a unidade de trabalho termina com o commit da transação do banco de dados – como definido pela opção de configuração da `thread` da classe `CurrentSessionContext`.

Você pode também querer carregar pessoas e eventos em diferentes unidades de trabalho. Ou modificar um objeto fora de uma `Session`, quando não se encontra no estado persistent (se já esteve neste estado anteriormente, chamamos esse estado de *detached*). Você pode até mesmo modificar uma coleção quando esta se encontrar no estado detached.

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session
            .createQuery("select p from Person p left join fetch p.events where p.id = :pid")
            .setParameter("pid", personId)
            .uniqueResult(); // Eager fetch the collection so we can use it detached

    Event anEvent = (Event) session.load(Event.class, eventId);

    session.getTransaction().commit();

    // End of first unit of work

    aPerson.getEvents().add(anEvent); // aPerson (and its collection) is detached

    // Begin second unit of work

    Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
    session2.beginTransaction();

    session2.update(aPerson); // Reattachment of aPerson

    session2.getTransaction().commit();
}
```

A chamada `update` cria um objeto persistent novamente, você poderia dizer que ele associa o objeto a uma nova unidade de trabalho, assim qualquer modificação que você faça neste objeto enquanto estiver no estado detached pode ser salvo no banco de dados. Isso inclui qualquer modificação (adição/exclusão) que você faça em

uma coleção da entidade deste objeto.

Bom, isso não foi muito usado na nossa situação, porém, é um importante conceito que você pode aplicar em seus aplicativos. Agora, complete este exercício adicionando uma nova ação ao método main( ) da classe `EventManager` e chame-o pela linha de comando. Se você precisar dos identificadores de uma pessoa ou evento – o método `save()` retorna estes identificadores (você poderá modificar alguns dos métodos anteriores para retornar aquele identificador):

```
else if (args[0].equals("addpersontoevent")) {
    Long eventId = mgr.createAndStoreEvent("My Event", new Date());
    Long personId = mgr.createAndStorePerson("Foo", "Bar");
    mgr.addPersonToEvent(personId, eventId);
    System.out.println("Added person " + personId + " to event " + eventId);
}
```

Este foi um exemplo de uma associação entre duas classes igualmente importantes, duas entidades. Como mencionado anteriormente, há outras classes e tipos dentro de um modelo típico, geralmente "menos importantes". Alguns você já viu, como um `int` ou uma `String`. Nós chamamos essas classes de *value types*, e suas instâncias *dependem* de uma entidade particular. As instâncias desses tipos não possuem sua própria identidade, nem são compartilhados entre entidades (duas pessoas não referenciam o mesmo objeto `firstname` mesmo se elas tenham o mesmo objeto firstname). Naturalmente, os value types não são apenas encontrados dentro da JDK (de fato, em um aplicativo Hibernate todas as classes JDK são consideradas como value types), mas você pode também criar suas classes como, por exemplo, `Address` ou `MonetaryAmount`.

Você também pode criar uma coleção de value types. Isso é conceitualmente muito diferente de uma coleção de referências para outras entidades, mas em Java parece ser quase a mesma coisa.

1.3.4. Coleção de valores

Nós adicionamos uma coleção de objetos de tipo de valores à entidade `Person`. Nós querermos armazenar endereços de e-mail, para isso utilizamos o tipo `String`, e a coleção novamente será um `Set`:

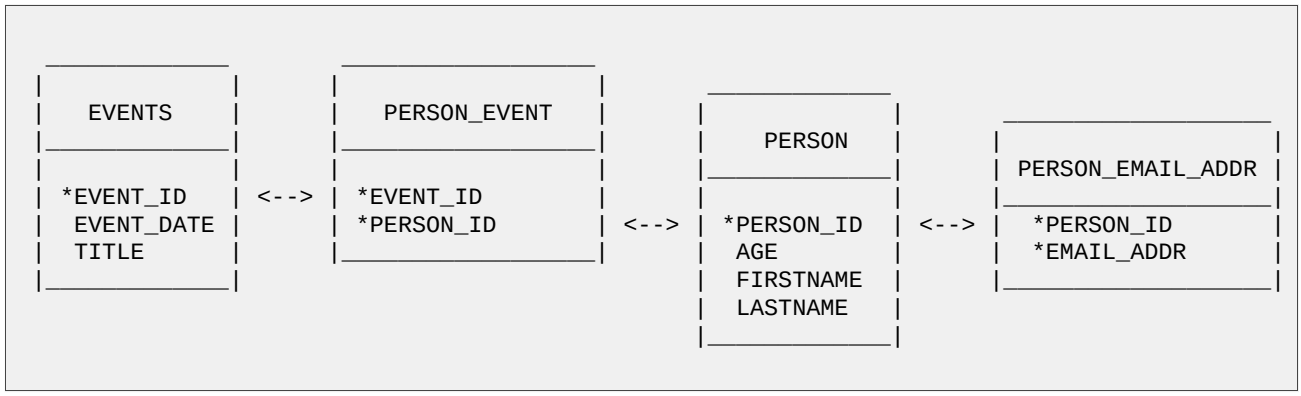
```
private Set emailAddresses = new HashSet();

public Set getEmailAddresses() {
    return emailAddresses;
}

public void setEmailAddresses(Set emailAddresses) {
    this.emailAddresses = emailAddresses;
}
```

O mapeamento deste `Set`:

```
<set name="emailAddresses" table="PERSON_EMAIL_ADDR">
    <key column="PERSON_ID"/>
    <element type="string" column="EMAIL_ADDR"/>
</set>
```

Comparando com o mapeamento anterior a diferença se encontra na parte `element`, que indica ao Hibernate que a coleção não contém referências à outra entidade, mas uma coleção de elementos do tipo `String` (a tag name em miniscula indica que se trata de um mapeamento do Hibernate para conversão de tipos). Mais uma vez, o atributo `table` do elemento `set` determina o nome da tabela para a coleção. O elemento `key` define o nome da coluna foreign key na tabela de coleção. O atributo `column` dentro do elemento `element` define o nome da coluna onde os valores da `String` serão armazenados.

Dê uma olhada no esquema atualizado:



Você pode observar que a primary key da tabela da coleção é de na verdade uma chave composta, usando ambas as colunas. Isso também implica que cada pessoa não pode ter endereços de e-mail duplicados, o que é exatamente a semântica que precisamos para um set em Java.

Você pode agora tentar adicionar elementos a essa coleção, do mesmo modo que fizemos anteriormente ligando pessoas e eventos. È o mesmo código em Java:

```
private void addEmailToPerson(Long personId, String emailAddress) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);

    // The getEmailAddresses() might trigger a lazy load of the collection
    aPerson.getEmailAddresses().add(emailAddress);

    session.getTransaction().commit();
}
```

Desta vez nós nós não usamos uma consulta *fetch* para inicializar a coleção. A chamada a seu método getter provocará um select adicional para inicializá-la, a partir dai, nós podemos adicionar um elemento nela . Monitore o log do SQL e tente otimizar usando eager fetch.

1.3.5. Associações bidirecionais

Agora iremos mapear uma associação bidirecional – fazendo a associação entre pessoas e eventos, de ambos os lados, em Java. Logicamente, o esquema do banco de dados não muda, nós continuamos tendo multiplicidades muitos-para-muitos. Um banco de dados é mais flexível do que uma linguagem de programação, ele não precisa de nenhuma direção de navegação – os dados podem ser acessados de qualquer forma.

Primeiramente, adicione uma coleção de participantes à classe `Event`:

```
private Set participants = new HashSet();

public Set getParticipants() {
    return participants;
}

public void setParticipants(Set participants) {
    this.participants = participants;
}
```

Agora mapeie este lado da associação em `Event.hbm.xml`.

```
<set name="participants" table="PERSON_EVENT" inverse="true">
    <key column="EVENT_ID"/>
```

```
    <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```

Como você pode ver, esse é um mapeamento normal usando `set` em ambos documentos de mapeamento. Observe que o nome das colunas em `key` e `many-to-many` estão trocados em ambos os documentos de mapeamento. A adição mais importante feita está no atributo `inverse="true"` no elemento set do mapeamento da coleção da classe `Event`.

Isso significa que o Hibernate deve pegar o outro lado – a classe `Person` – quando necessitar encontrar informação sobre a relação entre as duas entidades. Isso será muito mais facilmente compreendido quando você analisar como a relação bidirecional entre as entidades é criada.

1.3.6. Trabalhando com associações bidirecionais

Primeiro tenha em mente que o Hibernate não afeta a semântica normal do Java. Como nós criamos uma associação entre uma `Person` e um `Event` no exemplo unidirecional? Nós adicionamos uma instância de `Event`, da coleção de referências de eventos, a uma instância de `Person`. Então, obviamente, se nós quizermos que esta associação funcione bidirecionalmente, nós devemos fazer a mesma coisa do outro lado – adicionando uma referência de `Person` na coleção de um `Event`. Esse acerto de associações de ambos os lados é absolutamente necessário e você nunca deve esquecer de faze-lo.

Muitos desenvolvedores programam de maneira defensiva e criam métodos gerenciadores de associações que ajustam corretamente ambos os lados:

```
protected Set getEvents() {
    return events;
}

protected void setEvents(Set events) {
    this.events = events;
}

public void addToEvent(Event event) {
    this.getEvents().add(event);
    event.getParticipants().add(this);
}

public void removeFromEvent(Event event) {
    this.getEvents().remove(event);
    event.getParticipants().remove(this);
}
```

Observe que os métodos set e get da a coleção estão protegidos – isso permite que classes e subclasses do mesmo pacote continuem acessando os métodos, mas previne que qualquer outra classe, que não esteja no mesmo pacote, acesse a coleção diretamente. Você provavelmente deve fazer a mesma coisa para a coleção do outro lado.

E sobre o mapeamento do atributo `inverse`? Pra você, e para o Java, uma associação bidirecional é simplesmente o fato de ajustar corretamente as referências de ambos os lados. O Hibernate, entretanto não possui a informação necessária para adaptar corretamente os estados `INSERT` e `UPDATE` do SQL, e precisa de ajuda para manipular as propriedades das associações bidirecionais. Fazer um lado da associação com o atributo `inverse` instrui o Hibernate para basicamente ignora-lo, considerando-o uma *cópia* do outro lado. Isso é tudo o que é necessário para o Hibernate trabalhar com todas as possibilidades transformando um modelo de navegação bidirecional em esquema de banco de dados do SQL. As regras que você deve lembrar são claras: Todas as associações bidirecionais necessitam que um lado possua o atributo inverse. Em uma associação de um-para-muitos, o lado de "muitos" deve conter o atributo `inverse`, já em uma associação de muitos-para-muitos você pode usar qualquer lado, não há diferença.

Agora, vamos portar este exemplo para um pequeno aplicativo para internet.

1.4. EventManager um aplicativo para internet

Um aplicativo para internet do Hibernate usa uma `Session` e uma `Transaction` quase do mesmo modo que um aplicativo standalone. Entretanto, alguns patterns comuns são úteis. Nós agora criaremos um `EventManagerServlet`. Esse servlet lista todos os eventos salvos no banco de dados, e cria um formulário HTML para entrada de novos eventos.

1.4.1. Criando um servlet básico

Crie uma nova classe no seu diretório fonte, no pacote `events`:

```
package events;

// Imports

public class EventManagerServlet extends HttpServlet {

    // Servlet code
}
```

O servlet manuseia somente requisições `GET` do HTTP, portanto o método que iremos implementar é `doGet()`:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
        throws ServletException, IOException {

    SimpleDateFormat dateFormatter = new SimpleDateFormat("dd.MM.yyyy");

    try {
        // Begin unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().beginTransaction();

        // Process request and render page...

        // End unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().commit();

    } catch (Exception ex) {
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().rollback();
        throw new ServletException(ex);
    }

}
```

O pattern que estamos aplicando neste código é chamado *session-per-request*. Quando uma requisição chega ao servlet, uma nova `Session` do Hibernate é aberta através da primeira chamada para `getCurrentSession()` em `SessionFactory`. Então uma transação do banco de dados é inicializada - todo acesso a dados deve ocorrer dentro de uma transação, não importando se o dado é de leitura ou escrita. (nós não devemos usar o modo auto-commit em aplicações).

Não use uma nova `Session` do Hibernate para cada operação no banco de dados. Use uma `Session` do Hibernate no escopo da requisição toda. Use `getCurrentSession()`, de modo que seja limitado automaticamente à thread corrente do Java.

Agora, as possibilidades de ações de uma requisição serão processadas e uma resposta HTML será renderizada. Nós já vamos chegar nesta parte.

Finalmente, a unidade de trabalho termina quando o processamento e a renderização estão completos. Se ocorrer algum erro durante o processamento ou a renderização, uma exceção será lançada e a transação do banco de dados encerrada. Isso completa o pattern `session-per-request`. Em vez de usar código de demarcação de transação em todo servlet você pode também criar um filtro servlet. Dê uma olhada no site do Hibernate e do Wiki para maiores informações sobre esse pattern, chamado *Open Session in View*.

1.4.2. Processando e renderizando

Vamos implementar o processamento da requisição e a restituição da página HTML.

```
// Write HTML header
PrintWriter out = response.getWriter();
out.println("<html><head><title>Event Manager</title></head><body>");

// Handle actions
if ( "store".equals(request.getParameter("action")) ) {

    String eventTitle = request.getParameter("eventTitle");
    String eventDate = request.getParameter("eventDate");

    if ( "".equals(eventTitle) || "".equals(eventDate) ) {
        out.println("<b><i>Please enter event title and date.</i></b>");
    } else {
        createAndStoreEvent(eventTitle, dateFormatter.parse(eventDate));
        out.println("<b><i>Added event.</i></b>");
    }
}

// Print page
printEventForm(out);
listEvents(out, dateFormatter);

// Write HTML footer
out.println("</body></html>");
out.flush();
out.close();
```

O estilo de código acima, misturando linguagem HTML e Java não será funcional em um aplicativo mais complexo—tenha em mente que neste manual nós estamos apenas ilustrando conceitos básicos do Hibernate. O código imprime um cabeçalho HTML e um rodapé. Dentro desta página, é mostrado um formulário em HTML, para entrada de novos eventos, e uma lista de todos os eventos contidos no banco de dados. O primeiro método é trivial e apenas imprime uma página HTML:

```
private void printEventForm(PrintWriter out) {
    out.println("<h2>Add new event:</h2>");
    out.println("<form>");
    out.println("Title: <input name='eventTitle' length='50'/><br/>");
    out.println("Date (e.g. 24.12.2009): <input name='eventDate' length='10'/><br/>");
    out.println("<input type='submit' name='action' value='store'/>");
    out.println("</form>");
}
```

O método `listEvents()` usa a `Session` do Hibernate associada a thread atual para executar um query:

```
private void listEvents(PrintWriter out, SimpleDateFormat dateFormatter) {

    List result = HibernateUtil.getSessionFactory()
                    .getCurrentSession().createCriteria(Event.class).list();
    if (result.size() > 0) {
```

```
        out.println("<h2>Events in database:</h2>");
        out.println("<table border='1'>");
        out.println("<tr>");
        out.println("<th>Event title</th>");
        out.println("<th>Event date</th>");
        out.println("</tr>");
        for (Iterator it = result.iterator(); it.hasNext();) {
            Event event = (Event) it.next();
            out.println("<tr>");
            out.println("<td>" + event.getTitle() + "</td>");
            out.println("<td>" + dateFormatter.format(event.getDate()) + "</td>");
            out.println("</tr>");
        }
        out.println("</table>");
    }
}
```

Finalmente, a action `store` é passada pra o método `createAndStoreEvent()`, que também usa a `Session` da thread atual:

```
protected void createAndStoreEvent(String title, Date theDate) {
    Event theEvent = new Event();
    theEvent.setTitle(title);
    theEvent.setDate(theDate);

    HibernateUtil.getSessionFactory()
                    .getCurrentSession().save(theEvent);
}
```

Pronto, o servlet está completo. Uma requisição para o servlet será processada em uma `Session` e uma `Transaction` simples. Como anteriormente, no aplicativo standalone, o Hibernate pode automaticamente associar esses objetos a thread atual em execução. Isso possibilita a liberdade de você modelar seu código e acessar o método `SessionFactory` do jeito que achar melhor. Geralmente você irá usar um design mais sofisticado e mover o código de acesso a dados para dentro de objetos de acesso a dados (o pattern DAO). Leia o Hibernate Wiki para maiores exemplos.

1.4.3. Instalando e testando

Para fazer o deploy desta aplicação você tem que criar um arquivo para web, um WAR. Adicione o alvo Ant abaixo em seu `build.xml`:

```
<target name="war" depends="compile">
    <war destfile="hibernate-tutorial.war" webxml="web.xml">
        <lib dir="${librarydir}">
          <exclude name="jsdk*.jar"/>
        </lib>

        <classes dir="${targetdir}"/>
    </war>
</target>
```

Esta target cria um arquivo chamado `hibernate-tutorial.war` no diretório do seu projeto. Ele empacota todas as bibliotecas e o arquivo de descrição `web.xml`, o qual é esperado no diretório base do seu projeto:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xs

    <servlet>
        <servlet-name>Event Manager</servlet-name>
```

```
        <servlet-class>events.EventManagerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Event Manager</servlet-name>
        <url-pattern>/eventmanager</url-pattern>
    </servlet-mapping>
</web-app>
```

Antes de você compilar e fazer o deploy desta aplicação web, veja que uma biblioteca adicional é necessária: `jsdk.jar`. Esse é o Java servlet development kit, se você não possui esta biblioteca, faça seu download na página da Sun e copie-a para seu diretório de bibliotecas. Entretanto, será usado somente para a compilação e excluído do pacote WAR.

Para compilar e instalar execute `ant war` no seu diretório do projeto e copie o arquivo `hibernate-tutorial.war` para o diretório `webapp` do Tomcat. Se você não possui o Tomcat instalado faça o download e siga as instruções de instalação. Você não precisa modificar nenhuma configuração do Tomcat para rodar este aplicativo.

Uma vez feito o deploy e com Tomcat rodando, acesse o aplicativo em `http://localhost:8080/hibernate-tutorial/eventmanager`. Veja o log do Tomcat para observar a inicialização do Hibernate quando a primeira requisição chega ao servlet (o inicializador estático dentro de `HibernateUtil` é chamado) e para ter uma depuração detalhada se ocorrer alguma exceção.

1.5. Sumário

Este manual cobriu os princípios básicos para criação de uma aplicação simples do Hibernate e uma pequena aplicação web.

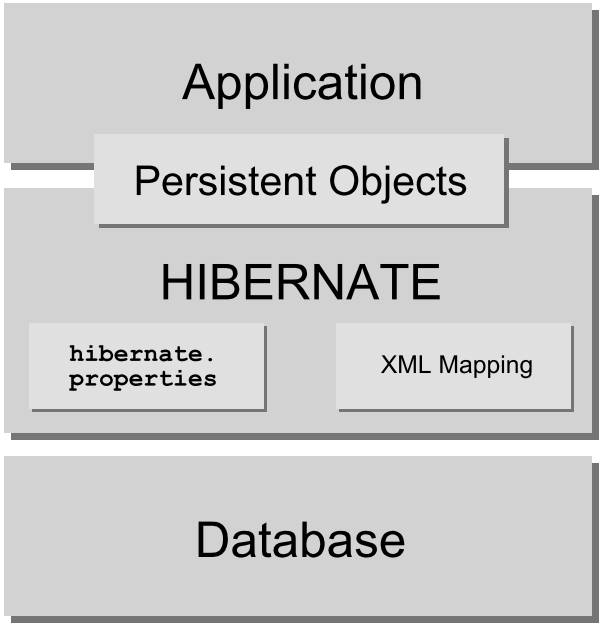
Se você já se sente seguro com o Hibernate, continue navegando na documentação de referência pelos tópicos que você achar mais interessante – os tópicos que geram mais perguntas são: processo de transação (Capítulo 11, *Transações e Concorrência*), uso da API (Capítulo 10, *Trabalhando com objetos*) e características de consulta (Seção 10.4, “Consultando”).

Não esqueça de visitar o site do Hibernate para obter mais tutoriais especializados.

Capítulo 2. Arquitetura

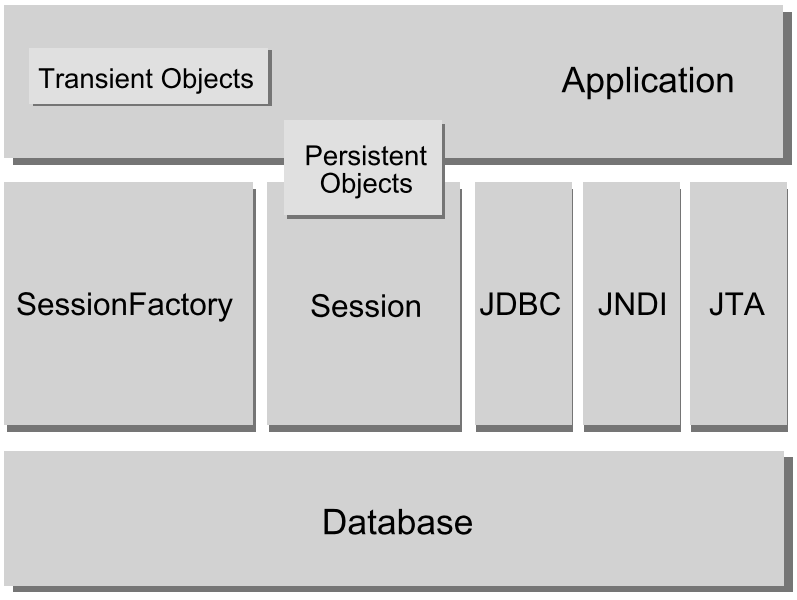
2.1. Visão Geral

Uma visão bem ampla da arquitetura do Hibernate:



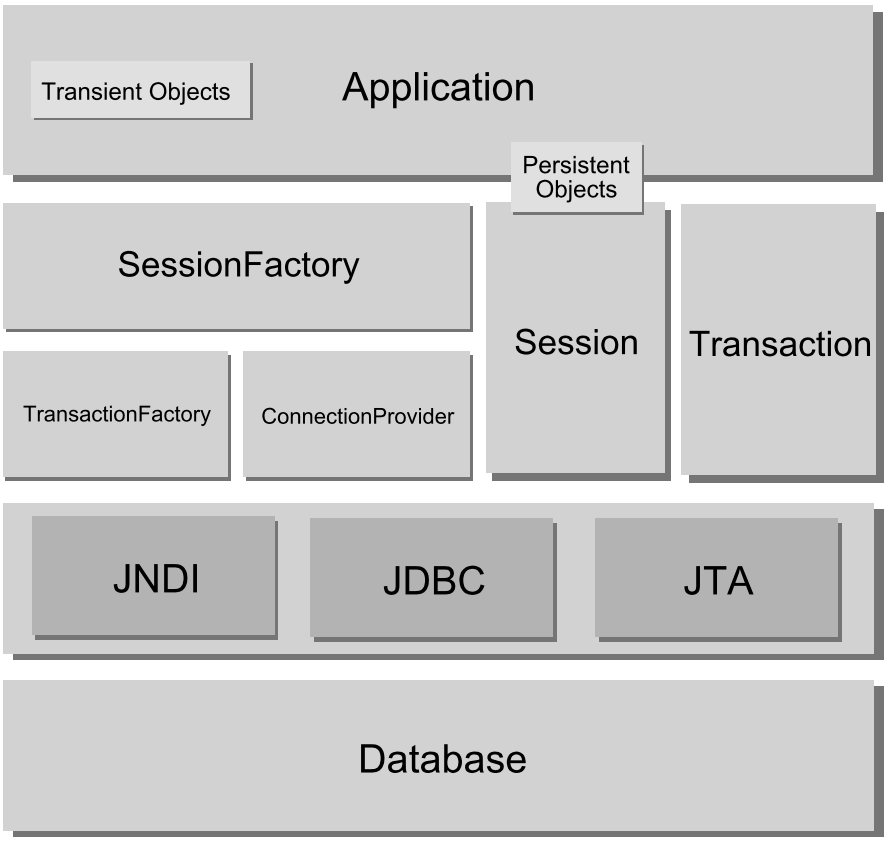
Esse diagrama mostra o Hibernate usando o banco de dados e a configuração de dados para prover persistência de serviços (e persistência de objetos) para o aplicativo.

Nós gostaríamos de mostrar uma visão mais detalhada da arquitetura em execução. Infelizmente, o Hibernate é muito flexível e suporta várias abordagens. Nós iremos mostrar os dois extremos. Na arquitetura mais simples o aplicativo fornece suas próprias conexões JDBC e gerencia suas transações. Esta abordagem usa o mínimo de subconjuntos das APIs do Hibernate:



A arquitetura "completa" abstrai a aplicação de ter de lidar diretamente com JDBC/JTA e APIs e deixa o Hiber-

nate tomar conta dos detalhes.



Algumas definições dos objetos do diagrama:

SessionFactory (`org.hibernate.SessionFactory`)
: Um cache threadsafe (imutáveis) composto de identidades compiladas para um único banco de dados. Uma fabrica para `Session` e um cliente de `ConnectionProvider`. Pode conter um cachê opcional de dados (segundo nível) reutilizáveis entre transações, no nível de processo ou cluster.

Session (`org.hibernate.Session`)
: Objeto single-threaded, de vida curta, representando uma conversação entre o aplicativo e o armazenamento persistente. Cria uma camada sobre uma conexão JDBC. É uma fabrica de `Transaction`. Possui um cachê obrigatório (primeiro nível) de objetos persistentes, usado para navegação no gráficos de objetos e pesquisa de objetos pelo identificador.

Objetos persistentes e coleções
: Objetos, de vida curta, single threaded contendo estado persistente e função de negócios. Esses podem ser JavaBeans/POJOs, onde única coisa especial sobre eles é que são associados a (exatamente uma) `Session`. Quando a `Session` é fechada, eles são separados e liberados para serem usados dentro de qualquer camada da aplicação (Ex. diretamente como data transfer objects de e para a camada de apresentação)

Objetos e coleções desatachados e transientes
: Instâncias de classes persistentes que ainda não estão associadas a uma `Session`. Eles podem ter sido instanciados pela aplicação e não persistido (ainda) ou eles foram instanciados por uma `Session` que foi encerrada.

Transaction (`org.hibernate.Transaction`)
: (Opcional) Objeto de vida curta, single threaded, usado pela aplicação para especificar unidades atômicas de trabalho. Abstrai o aplicativo de lidar diretamente com transações JDBC, JTA ou CORBA. Uma `Sessi-`

on pode, em alguns casos, iniciar várias Transactions. Entretanto, a demarcação da transação, mesmo utilizando API ou Transaction subjacentes, nunca é opcional!

ConnectionProvider (org.hibernate.connection.ConnectionProvider)
: (Opcional) Uma fábrica de (e combinações de) conexões JDBC. Abstrai a aplicação de lidar diretamente com Datasource ou DriverManager. Não exposto para a aplicação, mas pode ser implementado ou estendido pelo programador.

TransactionFactory (org.hibernate.TransactionFactory)
: (Opcional) Uma fábrica para instâncias de Transaction. Não exposta a aplicação, mas pode ser extendida/ implementada pelo programador.

Extension Interfaces
: O Hibernate oferece várias opções de interfaces estendidas que você pode implementar para customizar sua camada persistente. Veja a documentação da API para maiores detalhes.

Dada uma arquitetura simples, o aplicativo passa pelas APIs Transaction/TransactionFactory e/ou ConnectionProvider para se comunicar diretamente com a transação JTA ou JDBC.

2.2. Estados de instância

Uma instância de uma classe persistentes pode estar em um dos três diferentes estados, que são definidos respeitando um *contexto persistente*. O objeto Session do Hibernate é o contexto persistente:

transiente
: A instância não é, e nunca foi associada com nenhum contexto persistente. Não possui uma identidade persistente (valor da primary key).

persistente
: A instância está atualmente associada a um contexto persistente. Possui uma identidade persistente (valor da primary key) e, talvez, correspondente a um registro no banco de dados. Para um contexto persistente em particular, o Hibernate *garante* que a identidade persistente é equivalente a identidade Java (na localização em memória do objeto).

desatachado
: A instância foi associada com um contexto persistente, porém este contexto foi fechado, ou a instância foi serializada por outro processo. Possui uma identidade persistente, e, talvez, corresponda a um registro no banco de dados. Para instâncias desatachadas, o Hibernate não garante o relacionamento entre identidade persistente e identidade Java.

2.3. Integração JMX

JMX é padrão J2EE para manipulação de componentes Java. O Hibernate pode ser manipulado por um serviço JMX padrão. Nós fornecemos uma implementação do MBean na distribuição, org.hibernate.jmx.HibernateService.

Para um exemplo de como instalar o Hibernate como um serviço JMX em um servidor de aplicativo JBoss, por favor, consulte o manual do usuário do JBoss. No JBoss As, você poderá ver os benefícios de se fazer o deploy usando JMX:

- *Session Management:* O ciclo de vida de uma Session do Hibernate pode ser automaticamente conectado a

um escopo de transação JTA. Isso significa que você não precisará mais abrir e fechar manualmente uma `Session`, isso se torna trabalho para um interceptor EJB do JBoss . Você também não precisa se preocupar, nunca mais, com demarcação de transação em seu código (a não ser que você prefira escrever uma camada persistente portável, para isso, use a API opcional do Hibernate `Transaction`). Você deve chamar `HibernateContext` para acessar uma `Session`.

- *HAR deployment:*: Normalmente você faz o deploy de um serviço JMX do Hibernate usando um serviço descritor de deploy do JBoss (em um EAR e/ou arquivo SAR), que suporta todas as configurações usuais de uma `SessionFactory` do Hibernate. Entretanto, você ainda precisa nomear todos os seus arquivos de mapeamento no descritor de deploy. Se você decidir usar a forma opcional de deploy HAR, o JBoss irá automaticamente detectar todos os seus arquivos de mapeamento no seu arquivo HAR.

Consulte o manual do usuário do JBoss AS, para obter maiores informações sobre essas opções.

Outra opção disponível como um serviço JMX são as estatísticas de execução do Hibernate. Veja a Seção 3.4.6, “Estatísticas do Hibernate”.

2.4. Suporte JCA

O Hibernate pode também ser configurado como um conector JCA. Por favor, visite o website para maiores detalhes. Entretanto, note que o suporte JCA do Hibernate ainda é considerado experimental.

2.5. Sessões contextuais

Muitas aplicações que usam o Hibernate precisam de algum tipo de sessão "contextual", onde uma dada sessão é na verdade um escopo de um contexto. Entretanto, através de aplicações a definição sobre um contexto é geralmente diferente; e contextos diferentes definem escopos diferentes. Aplicações usando versões anteriores ao Hibernate 3.0 tendem a utilizar tanto sessões contextuais baseadas em `ThreadLocal`, classes utilitárias como `HibernateUtil`, ou utilizar frameworks de terceiros(como Spring ou Pico) que provê sessões contextuais baseadas em proxy.

A partir da versão 3.0.1, o Hibernate adicionou o método `SessionFactory.getCurrentSession()`. Inicialmente, este assume o uso de transações `JTA`, onde a transação `JTA` define tanto o escopo quanto o contexto de uma sessão atual. O time do Hibernate mantém este recurso, desenvolvendo as diversas implementações do `JTA TransactionManager`, a maioria (se não todos) aplicativos deveria utilizar o gerenciador de transações `JTA` sendo ou não instalados dentro de um container `J2EE`. Baseado neste recurso, você deveria sempre utilizar sessões contextuais baseadas em `JTA`.

Entretanto, na versão 3.1, o processo por trás do método `SessionFactory.getCurrentSession()` é agora plugavel. Com isso, uma nova interface (`org.hibernate.context.CurrentSessionContext`) e um novo parâmetro de configuração (`hibernate.current_session_context_class`) foram adicionados para possibilitar a compatibilidade do contexto e do escopo na definição de sessões correntes.

De uma olhada nos Javadocs sobre a interface `org.hibernate.context.CurrentSessionContext` para uma discussão detalhada. Ela define um único método, `currentSession()`, com o qual a implementação é responsável por rastrear a sessão contextual corrente. Por fora do "encapsulamento", o Hibernate possui duas implementações dessa interface.

- `org.hibernate.context.JTASessionContext` - As sessões correntes são rastreadas e recebem um escopo por uma transação `JTA`. O processamento aqui é exatamente igual ao antigo processo JTA. Consulte o Java-

doc para maiores detalhes.

- `org.hibernate.context.ThreadLocalSessionContext` - As sessões correntes são mantidas na thread de execução. Novamente, consulte o Javadoc para maiores detalhes.

- `org.hibernate.context.ManagedSessionContext` - as sessões atuais são mantidas na thread em execução. Entretanto, você é o responsável por fazer o bind o o unbind na instancia da `Session` com métodos estáticos desta classeclasse, essa abordagem nunca abre, limpa ou fecha a `Session` .

As duas primeiras implementações usam o modelo de programação "uma sessão – uma transação do banco de dados", também conhecida e usada como *sessão por requisição*. O começo e o fim de uma sessão Hibernate são definidos pela duração da transação do banco de dados. Se você usa marcação de transação de forma programática (por exemplo. em J2SE puro ou com JTA /UserTransaction/BMT), Nos recomendamos o uso da API Hibernate `Transaction` para separar o código de transação do seu código. Se você estiver usando um container EJB que tem suporte a CMT, os limites das transações são definidos de forma declarativa e você não precisa de qualquer operação de demarcação de transação ou sessão no seu código. Consulte Capítulo 11, *Transações e Concorrência* para mais informações e exemplos de código.

O parâmetro de configuração `hibernate.current_session_context_class` define que a implementação `org.hibernate.context.CurrentSessionContext` deve ser usada. Veja que para compatibilidade anterior, se este parâmetro de configuração não é determinado mas, um `org.hibernate.transaction.TransactionManagerLookup` é configurado, O Hibernate usará o `org.hibernate.context.JTASessionContext`. Tipicamente, o valor deste parâmetro seria apenas o nome de uma classe implementada pelo usuário. para as implementações out-of-the-box, entretanto, há três pequenos nomes correspondentes, "jta", "thread", e "managed".

Capítulo 3. Configuração

Devido ao fato de o Hibernate ser projetado para operar em vários ambientes diferentes, há um grande número de parâmetros de configuração. Felizmente, a maioria tem valores default lógicos e o Hibernate é distribuído com um arquivo `hibernate.properties` de exemplo no `etc/` que mostra várias opções. Apenas coloque o arquivo de exemplo no seu classpath e personalize-o.

3.1. 1.11 Configuração programática

Uma instância de `org.hibernate.cfg.Configuration` representa um conjunto inteiro de mapeamentos dos tipos Java da aplicação para um banco de dados SQL. O `Configuration` é usado para construir uma `SessionFactory` (imutável). Os mapeamentos são compilados a partir de arquivos de mapeamento XML.

Você pode obter uma instância `Configuration` instanciando- o diretamente e especificando documentos de mapeamento XML. Se os arquivos de mapeamento estiverem no classpath, use `addResource()`:

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

Uma alternativa (às vezes melhor) é especificar a classe mapeada, e permitir que o Hibernate encontre o documento de mapeamento para você:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Então o Hibernate procurará pelos arquivos de mapeamento chamados `/org/hibernate/auction/Item.hbm.xml` e `/org/hibernate/auction/Bid.hbm.xml` no classpath. Esta abordagem elimina qualquer nome de arquivo de difícil compreensão.

Uma `Configuration` também permite você especificar propriedades de configuração:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

Este não é o único caminho para passar as propriedades de configuração para o Hibernate. As várias opções incluem:

1. Passar uma instância de `java.util.Properties` para `Configuration.setProperties()`.
2. Colocar `hibernate.properties` no diretório raiz do classpath.
3. Determinar as propriedades do `System` usando `java -Dproperty=value`.
4. Incluir elementos `<property>` no `hibernate.cfg.xml` (discutido mais tarde).

`hibernate.properties` é o caminho mais facil se você quer começar rápidamente.

O `Configuration` é entendido como um objeto de inicialização, e é descartado uma vez que a `SessionFactory` esteja criada.

3.2. Obtendo uma SessionFactory

Quando todos os mapeamentos têm sido analisados pelo `Configuration`, a aplicação deve obter uma factory para as instâncias da `Session`. O objetivo desta factory é ser compartilhado por todas as threads da aplicação:

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Hibernate permite sua aplicação instanciar mais do que uma `SessionFactory`. Isto é útil se você estiver usando mais de um banco de dados.

3.3. Conexões JDBC

Normalmente, você desejará que a `SessionFactory` crie um pool de conexões JDBC para você. Se você seguir essa abordagem, a abertura de uma `Session` é tão simples quanto:

```
Session session = sessions.openSession(); // open a new Session
```

Assim que você fizer algo que necessitar de acesso ao banco de dados, uma conexão JDBC será obtida do pool.

Para esse trabalho, nós necessitamos passar algumas propriedades da conexão JDBC para o Hibernate. Todos os nomes de propriedades Hibernate e semânticas são definidas `org.hibernate.cfg.Environment`. Nós iremos descrever agora as mais importantes configurações da conexão JDBC.

O Hibernate obterá conexões(e pool) usando `java.sql.DriverManager` se você determinar as seguintes propriedades:

Tabela 3.1. Propriedades JDBC Hibernate

Nome da Propriedade	Propósito
`hibernate.connection.driver_class`	*Classe driver jdbc*
`hibernate.connection.url`	*URL jdbc*
`hibernate.connection.username`	*Usuário do banco de dados*
`hibernate.connection.password`	*Senha do usuário do banco de dados*
`hibernate.connection.pool_size`	*Número máximo de connecxões no pool*

O algoritmo de pool de conexões do próprio Hibernate entretanto é completamente rudimentar. A intenção dele e ajudar no inicio e *não para ser usado em um sistema de produção* ou até para testar desempenho. Você deveria usar uma ferramenta de pool de terceiros para conseguir melhor desempenho e estabilidade. Apenas especifique a propriedade `hibernate.connection.pool_size` com a definição do pool de conexões. Isto irá desligar o pool interno do Hibernate. Por exemplo, você pode gostar de usar o C3P0.

O C3P0 é um pool de conexão JDBC open source distribuído junto com o Hibernate no diretório `lib`. O Hibernate usará o `C3P0ConnectionProvider` para o pool de conexão se você configurar a propriedade `hibernate.c3p0.*`. Se você gostar de usar Proxool consulte ao pacote `hibernate.properties` e o web site do Hibernate para mais informações.

Eis um exemplo de arquivo `hibernate.properties` para C3P0:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

Para ser usado dentro de um servidor de aplicação, você deve configurar o Hibernate para obter conexões de um `Datasource` application server registrado no JNDI. Você necessitará especificar pelo menos uma das seguintes propriedades:

Tabela 3.2. Propriedades do Datasource do Hibernate

Nome da Propriedade	Propósito
`hibernate.connection.datasource`	*Nome datasource JNDI*
`hibernate.jndi.url`	*URL do fornecedor JNDI* (opcional)
`hibernate.jndi.class`	*Classe do JNDI* `InitialContextFactory` (opcional)
`hibernate.connection.username`	*Usuário do banco de dados* (opcional)
`hibernate.connection.password`	*Senha do usuário do banco de dados* (opcional)

Eis um exemplo de arquivo `hibernate.properties` para um servidor de aplicação provedor de datasources JNDI:

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

Conexões JDBC obtidas de um datasource JNDI automaticamente irão participar das transações gerenciadas pelo container no servidor de aplicação.

Arbitrariamente as propriedades de conexão podem ser acrescentandas ao "`hibernate.connnection`" ao nome da propriedade. Por exemplo, você deve especificar o `charSet` usando `hibernate.connection.charSet`.

Você pode definir sua própria estratégia de plugin para obter conexões JDBC implementando a interface `org.hibernate.connection.ConnectionProvider`. Você pode escolher uma implementação customizada setando `hibernate.connection.provider_class`.

3.4. Propriedades opcionais de configuração

Há um grande número de outras propriedades que controlam o comportamento do Hibernate em tempo de execução. Todos são opcionais e tem valores default lógicos.

Aviso: algumas destas propriedades são somente a "nível de sistema". Propriedades de nível de sistema somente podem ser configuradas via `java -Dproperty=value` ou `hibernate.properties`. Elas *não*podem ser configuradas por outras técnicas descritas abaixo.

Tabela 3.3. Propriedades de configuração do Hibernate

Nome da Propriedade	Propósito
`hibernate.dialect`	O nome da classe de um `Dialeto` que permite o Hibernate gerar SQL otimizado para um banco de dados relacional em particular. *Ex.* `full.classname.of.Dialect`
`hibernate.show_sql`	Escreve todas as instruções SQL no console. Esta é uma alternativa a configurar a categoria de log `org.hibernate.SQL` para `debug`. *Ex.* `true` | `false`
`hibernate.format_sql`	Imprime o SQL formatado no log e console. *f* `true` | `false`
`hibernate.default_schema`	Qualifica no sql gerado, os nome das tabelas sem qualificar com schena/tablespace dado *Ex.* `SCHEMA_NAME`
`hibernate.default_catalog`	Qualifica no sql gerado, os nome das tabelas sem qualificar com catálogo dado *Ex.* `CATALOG_NAME`
`hibernate.session_factory_name`	O `SessionFactory` sera associado automaticamente a este nome no JNDI depois de ter sido criado. *Ex.* `jndi/composite/name`
`hibernate.max_fetch_depth`	Estabelece a "profundidade" máxima para árvore outer join fetch para associações finais únicas(one-to-one, many-to-one). Um `0` desativa por default o uso de outer join em cosultas. *Ex.* Valores recomendados entre`0` e `3`
`hibernate.default_batch_fetch_size`	Determina um tamanho default para busca de associações em lotes do Hibernate *Ex.* Valores recomendados `4`, `8`, `16`
`hibernate.default_entity_mode`	Determina um modo default para representação de entidades para todas as sessões abertas desta `SessionFactory` *Ex.* `dynamic-map`, `dom4j`, `pojo`
`hibernate.order_updates`	Força o Hibernate a ordenar os updates SQL pelo valor da chave primária dos itens a serem atualizados. Isto resultará em menos deadlocks nas transações em sistemas altamente concorrente.

Nome da Propriedade	Propósito
	Ex. `true` | `false`
`hibernate.generate_statistics`	Se habilitado, o Hibernate coletará estatísticas úties para performance tuning dos bancos. *Ex.* `true` | `false`
`hibernate.use_identifer_rollback`	Se habilitado, propriedades identificadoras geradas serão zeradas para os valores default quando os objetos forem apagados. *Ex.* `true` | `false`
`hibernate.use_sql_comments`	Se habilitado, o Hibernate irá gerar comentários dentro do SQL, para facilitar o debugging, o valor default é `false`. *Ex.* `true` | `false`

Tabela 3.4. Propriedades JDBC e de Conexão do Hibernate

Nome da Propriedade	Propósito
`hibernate.jdbc.fetch_size`	Um valor maior que zero determina o tamanho do fetch do JDBC(chamadas `Statement.setFetchSize()`).
`hibernate.jdbc.batch_size`	Um valor maior que zero habilita uso de updates em lote do JDBC2 pelo Hibernate. *Ex.* valores recomentados entre `5` e `30`
`hibernate.jdbc.batch_versioned_data`	Configure esta propriedade como `true` se seu driver JDBC retorna o número correto de linhas no `executeBatch()` (É usualmente seguro deixar esta opção ligada). O Hibernate então irá usar DML em lote para versionar automaticamente dados. Valor default, `false` . *Ex.* `true` | `false`
`hibernate.jdbc.factory_class`	Escolhe um `Batcher` customizado. Muitas aplicações não irão necessitar desta propriedade de configuração *Ex.* `classname.of.Batcher`
`hibernate.jdbc.use_scrollable_resultset`	Habilita o uso de resultsets paginàveis do JDBC2 pelo Hibernate. Essa propriedade somente é necessaria quando se usa Conexeções JDBC providas pelo usuário, caso contrário o Hibernate usa os metadados da conexão. *Ex.* `true` | `false`

Nome da Propriedade	Propósito
hibernate.jdbc.use_streams_for_binary	Usa streams para escrever/ler tipos binary ou serializable para/a o JDBC (propriedade a nível de sistema). *Ex.* true | false
hibernate.jdbc.use_get_generated_keys	Possibilita o uso do PreparedStatement.getGeneratedKeys() em drivers JDBC3 para recuperar chaves geradas nativamente depois da inserção. Requer driver JDBC3+ e JRE1.4+, Configure como false se seu driver tem problemas com gerador de indentificadores Hibernate. Por default, tente determinar se o driver é capaz de usar metadados da conexão. *Ex.* true|false
hibernate.connection.provider_class	O nome da classe de um ConnectionProvider personalizado o qual proverá conexões JDBC para o Hibernate. *Ex.* classname.of.ConnectionProvider
hibernate.connection.isolation	Determina o nível de isolamento de uma transação JDBC. Verifique java.sql.Connection para valores siginificativos mas note que a maior parte dos bancos de dados não suportam todos os níveis de isolamento. *Ex.* 1, 2, 4, 8
hibernate.connection.autocommit	Habilita autocommit para conexões no pool JDBC(não recomendado). *Ex.* true | false
hibernate.connection.release_mode	Especifica quando o Hibernate deve liberar conexões JDBC. Por default, uma conexão JDBC é retida até que a sessão seja explicitamente fechada ou desconectada. Para um datasource de um servidor de aplicação JTA, você deve usar after_statement para forçar s liberação da conexões depois de todas as chamadas JDBC. Para uma conexão que não seja do tipo JTA, freqüentemente faz sentido liberar a conexão ao fim de cada transação, usando after_transaction. auto escolheremos after_statement para as estratégias de transaçãoes JTA e CMT e after_transaction para as estratégias de transação JDBC *Ex.* auto (default) | on_close | after_transaction | after_statement Veja que esta configuração só afeta as Sessions retornadas a partir de SessionFactory.openSession. Para Sessions obtidas através de SessionFac-

Nome da Propriedade	Propósito
	tory.getCurrentSession, a implementação de CurrentSessionContext deve ser configurada para controlar o modo de release de conexão para essas Sessions. Veja Seção 2.5, “Sessões contextuais”
hibernate.connection.<*propertyName*>	Passa a propriedade JDBC propertyName para DriverManager.getConnection().
hibernate.jndi.<*propertyName*>	Passar a propriedade propertyName para o InitialContextFactory JNDI.

Tabela 3.5. Propriedades de Cachê do Hibernate

Nome da Propriedade	Propósito
hibernate.cache.provider_class	O nome da classe de um CacheProvider customizado. *Ex.* classname.of.CacheProvider
hibernate.cache.use_minimal_puts	Otimiza operação de cachê de segundo nível para minimizar escritas, ao custo de leituras mais frequantes. Esta configuração é mais útil para cachês clusterizados e, no Hibernate3, é habilitado por default para implementações de cachê clusterizados. *Ex.* true|false
hibernate.cache.use_query_cache	Habilita a cache de consultas, Mesmo assim, consultas individuais ainda tem que ser habilitadas para o cache. *Ex.* true|false
hibernate.cache.use_second_level_cache	Pode ser usado para desabilitar o cache de segundo nível que é habilitado por default para classes que especifiquem <cache> no mapeamento. *Ex.* true|false
hibernate.cache.query_cache_factory	O nome de uma classe que implementa a interface QueryCache personalizada, por default, um StandardQueryCache criado automaticamente. *Ex.* classname.of.QueryCache
hibernate.cache.region_prefix	Um prefixo para usar nos nomes da área especial do cachê de segundo nível. *Ex.* prefix
hibernate.cache.use_structured_entries	Força o Hibernate a armazenar os dados no cachê de segundo nível em um formato mais legivel. *Ex.* true|false

Tabela 3.6. Propriedades de Transação do Hibernate

Nome da Propriedade	Propósito
`hibernate.transaction.factory_class`	O nome da clase de um a `TransactionFactory` para usar com API `Transaction` (por default JDBCTransactionFactory). *Ex.* `classname.of.TransactionFactory`
`jta.UserTransaction`	Um nome JNDI usado pelo `JTATransactionFactory` para obter uma `UserTransaction` JTA a partir do servidor de aplicação. *Ex.* `jndi/composite/name`
`hibernate.transaction.manager_lookup_class`	O nome da classe de um `TransactionManagerLookup` – requerido quando caching a nível JVM esta habilitado ou quando estivermos usando um generator hilo em um ambiente JTA. *Ex.* `classname.of.TransactionManagerLookup`
`hibernate.transaction.flush_before_completion`	Se habilitado, a sessão será automaticamente limpa antes da fase de conclusão da transação. É preferivel a gerência interna e automática do contexto da sessão, veja Seção 2.5, “Sessões contextuais” *Ex.* `true` | `false`
`hibernate.transaction.auto_close_session`	Se habilitado, a sessão será automaticamente fechada após a conclusão da transação. É preferivel a gerência interna e automática do contexto da sessão, veja Seção 2.5, “Sessões contextuais” *Ex.* `true` | `false`

Tabela 3.7. Propriedades Variadas

Nome da Propriedade	Propósito
`hibernate.current_session_context_class`	Fornece uma estratégia (personalizada) para extensão da `Session` "corrente". Veja Seção 2.5, “Sessões contextuais” para mais informação sobre estratégias internas. *Ex.* `jta` | `thread` | `managed` | `custom.Class`
`hibernate.query.factory_class`	Escolha a implementação de análise HQL. *Ex.* `org.hibernate.hql.ast.ASTQueryTranslatorFactory` or `org.hibernate.hql.classic.ClassicQueryTranslatorFactory`

Nome da Propriedade	Propósito
`hibernate.query.substitutions`	Mapeamento a partir de símbolos em consultas HQL para símbolos SQL(símbolos devem ser funções ou nome literais , por exemplo). *Ex.* `hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC`
`hibernate.hbm2ddl.auto`	Automaticamente valida ou exporta schema DDL para o banco de dados quando a `SessionFactory` é criada. Com `create-drop`, o schema do banco de dados será excluido quando a `create-drop` for fechada esplicitamente. *Ex.* `validate` | `update` | `create` | `create-drop`
`hibernate.cglib.use_reflection_optimizer`	Habilita o uso de CGLIB em vez de reflexão em tempo de execução (propriedade a nível de sistema). Reflexão pode algumas vezes ser ú til quando controlar erros, note que o Hibernate sempre irá requerer a CGLIB mesmo se você desligar o otimizador. Você não pode determinar esta propriedade no hibernate.cfg.xml. *Ex.* `true` | `false`

3.4.1. Dialetos SQL

Você deve sempre determinar a propriedade `hibernate.dialect` para a subclasse de `org.hibernate.dialect.Dialect` correta do seu banco de dados. Se você especificar um dialeto, o Hibernate usará valores defaults lógicos para as outras propriedades listadas abaixo, reduzindo o esforço de especificá-las manualmente.

Tabela 3.8. Hibernate SQL Dialects (`hibernate.dialect`)

RDBMS	Dialect
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
MySQL	`org.hibernate.dialect.MySQLDialect`
MySQL with InnoDB	`org.hibernate.dialect.MySQLInnoDBDialect`
MySQL with MyISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle (any version)	`org.hibernate.dialect.OracleDialect`
Oracle 9i/10g	`org.hibernate.dialect.Oracle9Dialect`
Sybase	`org.hibernate.dialect.SybaseDialect`

RDBMS	Dialect
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`org.hibernate.dialect.SQLServerDialect`
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

3.4.2. Recuperação por união externa (Outer Join Fetching)

Se seu banco de dados suporta recuperação por união externa (Outer Join Fetching) no estilo ANSI, Oracle ou Sybase, a recuperação por união externa (Outer Join Fetching) frequentemente aumentará o desempenho limitando o número de chamadas (round trips) ao banco de dados(ao custo de possivelmente mais trabalho desempenhado pelo próprio banco de dados). A recuperação por união externa (Outer Join Fetching) permite que um gráfico completo de objetos conectados por muitos-para-um, um-para-muitos, muitos-para-muitos e associações um-para-um sejam recuperadas em um unica instrução SQL SELECT.

A recuperação por união externa (Outer Join Fetching) pode ser desabilitado *globalmente* setando a propriedade `hibernate.max_fetch_depth` para `0`. Uma valor 1 ou maior habilita o outer join fetching para associações um-para-um e muitos-para-umos cujos quais tem sido mapeado com `fetch="join"`.

Veja Seção 19.1, “Estratégias de Fetching” para mais informações.

3.4.3. Fluxos Binários (Binary Streams)

O Oracle limita o tamanho de arrays de `byte` que pode ser passado para/de o driver JDBC. Se você desejar usar grandes instâncias de tipos `binary` ou `serializable`, você deve habilitar `hibernate.jdbc.use_streams_for_binary`. *Essa é uma configuração que só pode ser feita a nível de sistema.*

3.4.4. Cachê de segundo nível e consulta

As propriedades prefixadas pelo `hibernate.cache` permite você usar um sistema de cachê de segundo nível em um processo executado em cluster com o Hibernate. Veja Seção 19.2, “O Cache de segundo nível” para mais detalhes.

3.4.5. Substituições na Linguagem de Consulta

Você pode definir novos símbolos de consulta no Hibernate usando `hibernate.query.substitutions`. Por exemplo:

```
hibernate.query.substitutions true=1, false=0
```

Faria com que os símbolos `true` e `false` passasem a ser traduzidos para literais inteiro no SQL gerado.

```
hibernate.query.substitutions toLowercase=LOWER
```

permitirá você renomear a função `LOWER` no SQL.

3.4.6. Estatísticas do Hibernate

Se você habilitar `hibernate.generate_statistics`, o Hibernate exibirá um número de métricas bastante útil para o ajuste de sistema via `SessionFactory.getStatistics()`. O Hibernate pode até ser configurado para exibir essas estatísticas via JMX. Leia o Javadoc da interface `org.hibernate.stats` para mais informações.

3.5. Logging

Hibernate registra vários eventos usando Apache commons-logging.

O serviço commons-logging direcionará a saída para o Apache Log4j (se você incluir `log4j.jarr` no seu classpath) ou JDK1.4 logging(se estiver em uso JDK1.4 ou maior). Você pode fazer o download do Log4j a partir de `http://jakarta.apache.org`. Para usar Log4j você necessitará colocar um arquivo `log4j.properties` no seu classpath, um exemplo de arquivo de propriedades é distribuído com o Hibernate no diretório `src/`.

Nós recomendamos enfaticamente que você se familiarize-se com as mensagens de log do Hibernate. Uma parte do trabalho tem sido tornar o log do Hibernate o mais detalhado quanto possível, sem deixa-lo ilegível. É um dispositivo essencial de controle de erros. As categorias de log mais interessantes são as seguintes:

Tabela 3.9. Categorias de Log do Hibernate

Categoria	Função
`org.hibernate.SQL`	Registra todas as instruções SQL DML a medida que elas são executadas
`org.hibernate.type`	Registra todos os parâmetros JDBC
`org.hibernate.tool.hbm2ddl`	Registra todas as instruções SQL DDL a medida que elas são executadas
`org.hibernate.pretty`	Registra o estado de todas as entidades (máximo 20 entidades) associadas a session no momento da limpeza (flush).
`org.hibernate.cache`	Registra todas as atividades de cachê de segundo nível
`org.hibernate.transaction`	Registra atividades relacionada a transação
`org.hibernate.jdbc`	Registra todas as requisições de recursos JDBC
`org.hibernate.hql.ast.AST`	Registra instruções SQL e HQL durante a análise da consultas
`org.hibernate.secure`	Registra todas as requisições de autorização JAAS
`org.hibernate`	Registra tudo (uma parte das informações, mas muito útil para controle de er-

Categoria	Função
	ros)

Quando desenvolver aplicações com o Hibernate, você deve quase sempre trabalhar com `debug` para a categoria `org.hibernate.SQL`, ou, alternativamente, a com a propriedade `hibernate.show_sql` habilitada.

3.6. Implementando uma `NamingStrategy`

A interface `org.hibernate.cfg.NamingStrategy` permite você especificar um "padrão de nomes" para objetos do banco de dados e elementos schema.

Você deve criar regras para a geração automaticamente de identificadores do banco de dados a partir de identificadores Java ou para processar colunas "computadas" e nomes de tabelas informadas no arquivo de mapeamento para nomes "físicos" de tabelas e colunas. Esta característica ajuda a reduzir a verbosidade do documento de mapeamento, eliminando interferências repetitivas(`TBL_`prefixos, por exemplo). A estratégia default usada pelo Hibernate é completamente mínima.

Você pode especificar uma estratégia diferente ao chamar `Configuration.setNamingStrategy()` antes de adicionar os mapeamentos:

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

`org.hibernate.cfg.ImprovedNamingStrategy` é uma estratégia interna que pode ser um ponto de começo útil para algumas aplicações.

3.7. Arquivo de configuração XML

Uma maneira alternativa de configuração é especificar uma configuração completa em um arquivo chamado `hibernate.cfg.xml`. Este arquivo pode ser usado como um substituto para o arquivo `hibernate.properties` ou, se ambos estão presentes, sobrescrever propriedades.

Por default, espera-se que o arquivo XML de configuração estaja na raiz do seu `CLASSPATH`. Veja um exemplo:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory
        name="java:hibernate/SessionFactory">

        <!-- properties -->
        <property name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTransaction</property>
```

```
        <!-- mapping files -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- cache settings -->
        <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
        <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
        <collection-cache collection="org.hibernate.auction.Item.bids" usage="read-write"/>

    </session-factory>

</hibernate-configuration>
```

Como você pode ver, a vantagem deste enfoque é a externalização dos nomes dos arquivos de mapeamento para configuração. O `hibernate.cfg.xml` também é mais conveniente caso você tenha que ajustar o cache do Hibernate. Note que a escolha é sua em usar `hibernate.properties` ou `hibernate.cfg.xml`, ambos são equivalente, à exceção dos benefícios acima mencionados de se usar a sintaxe XML.

Com a configuração do XML, iniciar o Hibernate é então tão simples como

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

Você pode escolher um arquivo XML de configuração diferente usando

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

3.8. Integração com servidores de aplicação J2EE

O Hibernate tem os seguintes pontos da integração para o infraestrutura J2EE:

- *DataSources gerenciados pelo container*: O Hibernate pode usar conexões JDBC gerenciadas pelo Container e fornecidas pela JNDI. Geralmente, um `TransactionManager` compatível com JTA e um `ResourceManager` cuidam do gerenciamento da transação (CMT), especialmente em transações distribuídas manipuladas através de vários DataSources. Naturalmente, você também pode demarcar os limites das transações via programação (BMT) ou você poderia querer usar a API opcional do Hibernate `Transaction` para esta manter seu código portável.

- *Ligação (binding) automática a JNDI*: O Hibernate pode associar sua `SessionFactory` a JNDI depois de iniciada.

- *Ligação (binding) Session na JTA:* A `Session` do Hibernate pode ser associada automaticamente ao escopo das transações JTA. Simplesmente localizando a `SessionFactory` da JNDI e obtendo a `Session` corrente. Deixe o Hibernate cuidar da limpeza e encerramento da `Session` quando as transações JTA terminarem. A Demarcação de transação pode ser declarativa (CMT) ou via programação (BMT/Transação do usuário).

- *JMX deployment:* Se você usa um servidor de aplicações com suporte a JMX (ex. Jboss AS), você pode fazer a instalação do Hibernate como um Mbean controlado. Isto evita ter que iniciar uma linha de código para construir sua `SessionFactory` de uma `Configuration`. O container iniciará seu `HibernateService`, e idealmente também cuidará das dependências de serviços (DataSources, têm que estar disponíveis antes do Hibernate iniciar, etc.).

Dependendo de seu ambiente, você pode ter que ajustar a opção de configuração `hibernate.connection.aggressive_release` para verdadeiro (true), se seu servidor de aplicações lançar exeções d tipo "retenção de conecção".

3.8.1. Configuração de estratégia de transação

A API `Session` do Hibernate é independente de qualquer sistema de controle de transação em sua arquitetura. Se você deixar o Hibernate usar a JDBC diretamente, através de um pool de conexões, você pode inicializar e encerrar suas transações chamando a API JDBC. Se você rodar seu código em um servidor de aplicações J2EE, você poderá usar transações controladas por beans e chamar a API JTA e `UserTransaction` quando necessário.

Para manter seu código portável entre estes dois (e outros) ambientes, recomendamos a API Hibernate `Transaction`, que envolve e esconde o sistema subjacente. Você tem que especificar um classe construtora para `Transaction` instanciar ajustando a propriedade de configuração do `hibernate.transaction.factory_class`.

Existem três escolhas (internas) padrões:

`org.hibernate.transaction.JDBCTransactionFactory`
: delega as transações (JDBC) ao banco de dados (Padrão)

`org.hibernate.transaction.JTATransactionFactory`
: delega as transações a um container gerenciador se a transação existente estiver de acordo com contexto (ex: método bean sessão EJB), se não uma nova transação é iniciada e uma transação controlada por um bean é usada.

`org.hibernate.transaction.CMTTransactionFactory`
: delega para um container gerenciador de transações JTA

Você também pode definir suas próprias estratégias de transação (para um serviço de transação CORBA por exemplo).

Algumas características no Hibernate (ex., o cache de segundo nível, sessões contextuais com JTA, etc.) requerem acesso a JTA `TransactionManager` em um ambiente controlado. Em um servidor de aplicação você tem que especificar como o Hibernate pode obter uma referência para a `TransactionManager`, pois o J2EE não padroniza um mecanismo simples :

Tabela 3.10. Gerenciadores de transações JTA

Transaction Factory	Application Server
`org.hibernate.transaction.JBossTransactionManagerLookup`	JBoss
`org.hibernate.transaction.WeblogicTransactionManagerLookup`	Weblogic
`org.hibernate.transaction.WebSphereTransactionManagerLookup`	WebSphere
`org.hibernate.transaction.WebSphereExtendedJTATransactionLookup`	WebSphere 6
`org.hibernate.transaction.OrionTransactionManagerLookup`	Orion
`org.hibernate.transaction.ResinTransactionManagerLookup`	Resin
`org.hibernate.transaction.JOTMTransactionManagerLookup`	JOTM
`org.hibernate.transaction.JOnASTransactionManagerLookup`	JOnAS

Transaction Factory	Application Server
org.hibernate.transaction.JRun4TransactionManagerLookup	JRun4
org.hibernate.transaction.BESTransactionManagerLookup	Borland ES

3.8.2. SessionFactory associada a JNDI

Uma SessionFactory do Hibernate associada a JNDI pode simplificar a localização da fabrica e a criação de novas Sessions. Veja que isto não está relacionado a um Datasource ligado a JNDI, simplemente ambos usam o mesmo registro!

Se você desejar ter uma SessionFactory associada a um namespace JNDI, especifique um nome (Ex. java:hibernate/SessionFactory) usando a propriedade hibernate.session_factory_name. Se esta propriedade for omitida, a SessionFactory não será associada a JNDI. (Isto é especialmente útil em ambientes com implementação padrão de um JNDI somente de leitura, por exemplo Tomcat.)

Ao se associar a SessionFactory a JNDI, o Hibernate usará os valores de hibernate.jndi.url, hibernate.jndi.class para instanciar o contexto inicial. Se eles não são especificados, o valor default InitialContext será usado.

O Hibernate tentará colocará automaticamente a SessionFactory no JNDI depois que você chamar cfg.buildSessionFactory(). Isto significa que você terá uma chamada desta no inicio do seu código (ou classe de utilitaria) em sua aplicação, a menos que você use desenvolva usando JMX com o HibernateService (discutido mais adiante).

Se você usa uma SessionFactory JNDI, um EJB ou qualquer outra classe para obter SessionFactory usando o lookup JNDI.

Nós recomendamos que você associe o SessionFactory a um JNDI em um ambiente gerenciado e use um singleton static caso contrário. Para proteger seu código de aplicação destes detalhes, nós recomendamos também esconder o código lookup para uma SessionFactory em uma classe helper, por exemplo HibernateUtil.getSessionFactory(). Veja que tal classe também é um modo conveniente de iniciar o Hibernate. veja o capítulo 1.

3.8.3. Administração de contexto de Sessão atual com JTA

O modo mais fácil para controlar Sessions e transações é a administração automática da Session "atual" pelo Hibernate. Veja Seção 2.5, “Sessões contextuais”. Usando o contexto "jta" da sessão, se não existe nemhuma Session do Hibernate associada a uma transação JTA , a a primeira coisa a se fazer é associar com uma transação JTA na primeira vez você chamar sessionFactory.getCurrentSession(). As Sessions devem ser carregadas através de getCurrentSession() no contexto "jta" para setar para executar o flush automaticamente antes da completar a transação, e depois que a transação estiver completa, fechar e liberar explicitamente as conexões JDBC depois de cada declaração. Isto permite que as Sessions sejam administradas pelo ciclo de vida da transação JTA com a qual estiver associado, mantendo seu código livre de tais preocupações de administração. Seu código pode usar a JTA via programação através UserTransaction, ou (recomendado para código portátil) usar a API Transaction do Hibernate para setar os limites de transação. Se você estiver trabalhando em um container EJB, demarcação de transação declarativa com CMT é preferida.

3.8.4. Deployment JMX

A linha cfg.buildSessionFactory() ainda tem que ser executadoa em algum lugar para recupear a Session-

`Factory` no JNDI. Você pode fazer isto em um bloco de initialização `static` (como em `HibernateUtil`) ou fazer o deploy do Hibernate como um *serviço administrado*.

O Hibernate é distribuído com `org.hibernate.jmx.HibernateService` para desenvolvimento de aplicações em um servidor com suporte JMX, como o JBoss As. A instalação e configuração é específica de cada fornecedor. Aqui é um exemplo de `jboss-service.xml` para JBoss 4.0.x:

```
<?xml version="1.0"?>
<server>

<mbean code="org.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory,name=HibernateFactory">

    <!-- Required services -->
    <depends>jboss.jca:service=RARDeployer</depends>
    <depends>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- Bind the Hibernate service to JNDI -->
    <attribute name="JndiName">java:/hibernate/SessionFactory</attribute>

    <!-- Datasource settings -->
    <attribute name="Datasource">java:HsqlDS</attribute>
    <attribute name="Dialect">org.hibernate.dialect.HSQLDialect</attribute>

    <!-- Transaction integration -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</attribute>
    <attribute name="FlushBeforeCompletionEnabled">true</attribute>
    <attribute name="AutoCloseSessionEnabled">true</attribute>

    <!-- Fetching options -->
    <attribute name="MaximumFetchDepth">5</attribute>

    <!-- Second-level caching -->
    <attribute name="SecondLevelCacheEnabled">true</attribute>
    <attribute name="CacheProviderClass">org.hibernate.cache.EhCacheProvider</attribute>
    <attribute name="QueryCacheEnabled">true</attribute>

    <!-- Logging -->
    <attribute name="ShowSqlEnabled">true</attribute>

    <!-- Mapping files -->
    <attribute name="MapResources">auction/Item.hbm.xml,auction/Category.hbm.xml</attribute>

</mbean>

</server>
```

Este arquivo é colocado em um diretório chamado `META-INF` e empacotado em um arquivo JAR com a extensão `.sar` (arquivo de serviço). Você também precisa empacotar o Hibernate, e as bibliotecas de terceiros exigidas, suas classes persistentes compiladas, bem como seus arquivos de mapeamentono mesmo arquivo. Seus enterprise beans (normalmente beans de sessão) podem ser mantidos em seus próprios arquivo JAR , mas você pode incluir estes arquivo EJB JAR no arquivo de serviço principal para ter um única unidade de (hot-)deploy. Consulte a documentação do o JBoss AS para mais informação sobre seviços JMX e desenvolvimento EJB.

Capítulo 4. Classes persistentes

Classes persistentes são classes em uma aplicação que implementaM as entidades do problema do negocio (por exemplo Cliente e Encomenda em uma aplicação de E-comerce). Nem todas as instancias de uma classe persistente estão no estado persistente - um instancia pode ser passageira ou destacada.

O Hibernate trabalha melhor se estas classes seguirem algumas regras simples, também conhecidas como modelo de programação POJO (Plain Old Java Object). Porém, nenhuma destas regras é obrigatória. Realmente, O Hibernate3 presume muito pouco sobre a natureza de seus objetos persistentes. Você pode expressar um modelo de domínio de outros modos: usando arvores de instancias de `Map`, por exemplo.

4.1. Um exemplo simples de POJO

A maioria que aplicações de Java requerem uma classe persistente que represente felinos.

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier

    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
```

```
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
            kitten.setMother(this);
        kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

Há quatro regras principais a se seguir:

4.1.1. Implementar um constructor sem argumentos

`Cat` tem um constructor sem argumentos. Todas a classes persistentes têm que ter um constructor default (que não precisa ser público) de forma que o Hibernate consiga criar uma nova instancia usando `Constructor.newInstance()`. Nós recomendamos enfaticamente ter um constructor default com pelo menos visibilidade de *package* para a geração proxy em runtime dentro do Hibernate.

4.1.2. Garanta a existencia de uma propriedade identificadora (opcional)

`Cat` tem uma propriedade chamada `id`. Esta propriedade mapeia a coluna primary key na tabela. A propriedade poderia ter qualquer nome, e seu tipo poderia ser qualquer tipo primitivo, qualquer "wrapper" para tipos primitivos, `java.lang.String` ou `java.util.Date`. (Se sua tabela de banco de dados legado tiver chaves compostas, você pode até usar uma classe definida pelo usuario, com propriedades destes tipos - veja a seção identificadores compostos abaixo.)

A propriedade de identificadora é estritamente opcional. Você pode escolher não implementa-la e deixar que Hibernate gerencie internamente os identificadores dos objetos. Sem dúvida, nós não recomendamos isso.

Na realidade, algumas funcionalidades só estão disponíveis a classes que declaram uma propriedade identificadora:

- Reassociação transitiva para objetos destacados (atualização de cascata ou merge em cascata) - veja Seção Seção 10.11, “Persistência transitiva”
- `Session.saveOrUpdate()`
- `Session.merge()`

Nós recomendamos que você declare propriedades identificadoras nomeadas de forma constantemente nas clas-

ses persistentes. Nós também recomendamos que você use tipos que possam ser nulos (ou seja, tipos não primitivos).

4.1.3. Prefira classes que não sejam marcadas como final(opcional)

Uma apecto central do Hibernate, os *proxies*, dependem de que as classes persistentes não sejam finais, ou que sejam a implementação de uma interface que declara todos os métodos públicos.

Você pode persistir classes `finais` que não implementam uma interface do Hibernate, mas você não poderá usar proxies para recuperação de associações lazy - o que limitará suas opções para a melhoria da perfomance.

Você deve evitar também declarar métodos como `public final` nas classes que não sejam finais. Se você quiser usar uma classe com um método `public final`, você deve desabilitar explicitamente o uso de proxies setando `lazy="false"`.

4.1.4. Declare metodos accessores e modificadores para os campos persistentes (opcional)

`Cat` declara métodos accessores para todos seus campos persistentes. Muitas outras ferramentas de ORM persistem variáveis de instancias diretamente. Nós acreditamos que é melhor prover uma indireção entre o schema relacional e a estrutura interna de classes. Por default, o Hibernate persiste propriedades no estilo JavaBeans, e reconhecem nomes de métodos na forma `getFoo`, `isFoo` e `setFoo`. Você pode trocar o acesso direto a campo por propriedades particulares, se precisar.

Propriedades *não* precisam ser declaradas com públicas - O Hibernate pode persistir uma propriedade com um par get / set `protected` ou `private`.

4.2. Implementando Herança

Uma subclasse também tem que obedecer a primeira e segunda regra. Ela herda sua propriedade identificadora da superclasse, `Cat`.

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }
        protected void setName(String name) {
                this.name=name;
        }
}
```

4.3. Implementando `equals()` e `hashCode()`

Você precisa sobreescrever os métodos `equals()` e `hashCode()` se você

- Pretente inserir instancias de classes persistentes em um `Set` (o modo recomendado para representar associações multivaloradas) *e*
- Pretente usar reassociação de instancias destacadas

O Hibernate garante a equivalência de identidade persistente (database row) e a identidade de Java apenas dentro do escopo de uma sessão em particular. De modo que no momento que misturamos instancias recuperadas em sessões difrentes, devemos implementar `equals()` e `hashCode()` se quisermos ter uma semantica significante dento dos `Sets`

A maneira mais obvia é implementar `equals()`/`hashCode()` comparando o valor de identificador de ambos os objetos. Se o valor for o mesmo, ambos devem ser a mesma linha no banco de dados, então, eles são iguais (se forem adicionado ambos a um `Set`, nós teremos só um elemento no `Set`). Infelizmente, nós não podemos usar a abordagem com identificadores gerados! O Hibernate somentre atribuirá valores de identificador a objetos que estiverem persistentes, um instancia recentemente criada não terá valor de identificador! Além disso, se uma instancia não esta salva e atualmente inserida em um `Set`, salva-la atribuiria um valor de identificador ao objeto. Se `equals()` e `hashCode()` estão baseado no valor de identificador, o código hash poderia mudar, quebrando o contrato do `Set`. Veja o site do Hibernate para uma discussão completa deste problema. Veja que isto não é uma caracteristica do Hibernate, mas uma semântica normal do Java para identidade de objeto e igualdade.

Nós recomendamos implementar `equals()` e `hashCode()` usando *Igualdade de chave de négocio*. Igualdade de chave de négocio siginifica que o método `equals()` compara só as propriedades que formam a chave de négocio, uma chave que identificaria nossa instancia no real mundo (uma chave de candidata *natural*):

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return false;
        if ( !cat.getMother().equals( getMother() ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

Veja que uma chave de negocio não tem que ser tão sólida quanto uma chave primaria candidata no banco de dados (veja Seção 11.1.3, “Considerando a identidade do objeto”). As propriedades imutáveis ou únicas são normalmente boas candidatas para uma chave de negocio.

4.4. Modelos dinâmicos

Veja que as características seguintes atualmente são consideradas experimentais e podem mudar em um futuro próximo.

Entidades persistentes não necessariamente têm que ser representadas como classes de POJO ou como objetos JavaBeans em tempo de execução. O hibernate suporta modelos dinâmicos (usando `Maps` de `Map` em tempo de execução) e a representação de entidades como árvores DOM4J. Com esta aboradagem, você não escreve classes persistentes, apenas arquivos de mapeamento.

Por default, o Hibernate trabalha em modo POJO normal. Você pode setar o modo de representação de entidade

para uma SessionFactory em particular usando aopção de configuração default_entity_mode (veja Tabela 3.3, “Propriedades de configuração do Hibernate”).

Os exemplos seguintes demonstram a representação que usando Maps. Primeiro, no arquivo de mapeamento, deve ser declarado em um entity-name vez de (ou além de) um nome de classe:

```
<hibernate-mapping>

    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
        </id>

        <property name="name"
            column="NAME"
            type="string"/>

        <property name="address"
            column="ADDRESS"
            type="string"/>

        <many-to-one name="organization"
            column="ORGANIZATION_ID"
            class="Organization"/>

        <bag name="orders"
            inverse="true"
            lazy="false"
            cascade="all">
            <key column="CUSTOMER_ID"/>
            <one-to-many class="Order"/>
        </bag>

    </class>

</hibernate-mapping>
```

Veja que embora as associações sejam declaradas usando nomes de classe alvo, o tipo alvo de uma associação também pode ser uma entidade dinâmica em vez de um POJO.

Depois de setar para dynamic-map o modo default de entidade para a SessionFactory, nós podemos trabalhar em tempo de execução com Maps de Maps:

```
Session s = openSession();
Transaction tx = s.beginTransaction();
Session s = openSession();

// Create a customer
Map david = new HashMap();
david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
s.save("Organization", foobar);

tx.commit();
s.close();
```

As vantagens do mapeamento dinâmico são, o tempo reduzido do ciclo de prototipação sem a necessidade de implementação de classes de entidade. Porém, você perde a verificação de tipo em tempo de compilação e muito provávelmente terá muitas exceções em tempo de execução. Graças ao Hibernate, o schema de banco de dados pode ser facilmente melhorado e normalizado, permitindo criar uma implementação apropriada do modelo de domínio mais tarde.

Modos de representação de entidade também podem ser configurados para a sessão `Session`:

```
Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession
```

Veja que a chamada a `getSession()` usando `EntityMode` está na API da `Session`, e não o `SessionFactory`. Desta forma, a nova `Session` compartilha a conexão de JDBC subjacente, transações, e outras informações de contexto. Isto significa você não tem que chamar `flush()` e `close()` na `Session` secundária, e também deixa a transação e a conexão por conta da unidade primária de trabalho.

Podem ser achadas mais informações sobre as capacidades de representação XML em Capítulo 18, *Mapeamento XML*.

4.5. Tuplas

A interface`org.hibernate.tuple.Tuplizer`, e suas sub-interfaces, são responsáveis por administrar uma representação particular de um pedaço de dados, determinado a representação `org.hibernate.EntityMode`. Se um determinado porção de dados é visto de como uma estrutura de dados, então uma tupla é a coisa que sabe criar tal estrutura de dados e como extrair valores da mesma e com injetar valores em tal estrutura de dados. Por exemplo, para o modo de entidadePOJO, a tupla correpondente sabe como criar o POJO por seu constructor e como acessar as propriedades de POJO usando os metodos accessores definido. Há dois tipos de Tuplas de alto-nível , representados pelas interfaces `org.hibernate.tuple.EntityTuplizer` e `org.hibernate.tuple.component.ComponentTuplizer`. `EntityTuplizer` são responsáveis para administrar os supracitados contratos com respeito a entidades, enquanto `ComponentTuplizers` faz o mesmo para componentes.

Usuários também podem criar suas próprias tuplas. Talvez você queria que outra implementação de `java.util.Map` que não `java.util.HashMap` seja usada para dynamic-map e entity-mode; ou talvez você precise definir uma estratégia diferente de geração de proxy que o usada por default. Ambas seriam possivel definindo uma implementação customizada de tuplas. As definições de Tuplas são associadas à entidades ou componentes que devem ser administrados. Voltando para o exemplo de nossa entidade de cliente:

```
<hibernate-mapping>
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
```

```
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>


public class CustomMapTuplizerImpl
        extends org.hibernate.tuple.entity.DynamicMapEntityTuplizer {
    // override the buildInstantiator() method to plug in our custom map...
    protected final Instantiator buildInstantiator(
            org.hibernate.mapping.PersistentClass mappingInfo) {
        return new CustomMapInstantiator( mappingInfo );
    }

    private static final class CustomMapInstantiator
            extends org.hibernate.tuple.DynamicMapInstantitor {
        // override the generateMap() method to return our custom map...
            protected final Map generateMap() {
                    return new CustomMap();
            }
    }
}
```

TODO: Document user-extension framework in the property and proxy packages TODO: Documentar o framework extensão de usuário nas propriedade e nos pacotes de proxyes

Capítulo 5. Mapeamento O/R Bassico

5.1. Declaração de mapeamento

Os mapeamentos objeto/elational são normalmente definidos em documentos XML. O documento de mapeamento é projetado para ser legível e editável a mão. A linguagem de mapeamento é centrada em Java, significando que os mapeamento são construídos ao redor declarações de classe persistentes, não em declarações de tabelas.

Veja que, embora muitos usuários do Hibernate preferem escrever o XML à mão, Existem várias ferramentas para gerar o documento de mapeamento, inclusive XDoclet, Middlegen e AndroMDA.

Iremos começar com um exemplo de mapeamento:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat"
            table="cats"
            discriminator-value="C">

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

                <discriminator column="subclass"
                     type="character"/>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>

                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>

                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>

                <subclass name="DomesticCat"
```

```
                    discriminator-value="D">

                        <property name="name"
                            type="string"/>

                </subclass>

        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

Discutiremos agora o conteúdo deste documento de mapeamento. Iremos descrever apenas os elementos do documento e atributos que são utilizados pelo Hibernate em tempo de execução. O documento de mapeamento também contém alguns atributos adicionais e opcionais que afetam os esquemas de banco de dados exportados pela ferramenta de exportação de esquemas. (Por exemplo, o atributo `not-null`).

5.1.1. Doctype

Todos os mapeamentos de XML devem declarar o doctype exibido. O DTD atual pode ser encontrado na URL abaixo, no diretório `hibernate-x.x.x/src/org/ hibernate` ou no `hibernate3.jar`. O Hibernate sempre irá procurar pelo DTD inicialmente no seu classpath. Se você tentar localizar o DTD usando uma conexão de internet, compare a declaração do DTD com o conteúdo do seu classpath

5.1.1.1. EntityResolver

Como mencionado previamente, o Hibernate tentará solucionar DTDs em seu classpath primeiro. A maneira na qual faz é registrando um implementação customizada de `org.xml.sax.EntityResolver` com o SAXReader usa para ler nos arquivos de xml. Este `EntityResolver` customizado reconhece dois namespaces de systemId diferente.

- O `namespace do Hibernate` é reconhecido sempre que o resolver encontra um systemId que começa com `http://hibernate.sourceforge.net/`; o resolver tenta solucionar estas entidades pelo classlaoder que carregou as classes do Hibernate.
- Um `namespace de usuário` é reconhecido sempre que resolver encontra um systemId que usa um `classpath://` protocolo de URL; o resolver tentará solucionar estas entidades por (1) o classloader de contexto da thread atual e (2) o classloader que carrega as classes do Hiberante.

Um exemplo de utilização de namespace de usuário

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" [
    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">
]>

<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
    <class>
    &types;
</hibernate-mapping>
```

Onde `types.xml` é um recurso no pacote de `your.domain` e contém um Seção 5.2.3, “Tipos de valores personalizados” customizado"

5.1.2. Mapeamento Hiberante

Este elemento tem diversos atributos opcionais. Os atributos `schema` e `catalog` especificam que tabelas referenciadas neste mapeamento pertencem ao esquema e/ou ao catalogo nomeado. Se especificados, os nomes das tabelas irão ser qualificados no schema ou catalog dado. Se não, os nomes das tabelas não serão qualificados. O atributo `default-cascade` especifica qual estilo de cascata será assumido pelas propriedades e coleções que não especificarm um atributo `cascade`. O atributo `auto-import` nos deixa utilizar nomes de classes não qualificados na linguagem de consulta, por default.

```
<hibernate-mapping
         schema="schemaName"                          (1)
         catalog="catalogName"                        (2)
         default-cascade="cascade_style"              (3)
         default-access="field|property|ClassName"    (4)
         default-lazy="true|false"                    (5)
         auto-import="true|false"                     (6)
         package="package.name"                       (7)
 />
```

(1) `schema` (opcional): O nome do esquema do banco de dados.

(2) `catalog` (opcional): O nome do catálogo do banco de dados.

(3) `default-cascade` (opcional – valor default `nenhum`): Um estilo cascata default.

(4) `default-access` (opcional – valor default `property`): A estratégia que o Hibernate deve utilizar para acessar todas as propridades. Pode ser uma implementação própria de `PropertyAccessor`.

(5) `default-lazy` (opcional - valor default `true`): O valor default para atributos `lazy` da classe e dos mapeamentos de coleções.

(6) `auto-import` (opcional - valr default`true`): Especifica se nós podemos usar nomes de classess não qualificados (das classes deste mapeamento) na linguagem de consulta.

(7) `package` (opcional): Especifica um prefixo da package para assumir para nomes de classes não qualificadas no documento de mapeamento.

Se você tem duas classes persistentes com o mesmo nome (não qualificadas), você deve setar `auto-import="false"`. O Hibernate irá gerar uma exceção se você tentar setar duas classes para o mesmo nome "importado".

Observe que o elemento `hibernate-mapping` permite a você aninhar diversos mapeamentos de `<class>` persistentes, como mostrado abaixo. Entretanto, é uma boa prática (e esperado por algumas ferramentas)o mapeamento de apenas uma classe persistente simples (ou uma hierarquia de classes simples) em um arquivo de mapeamento e nomea-la após a superclasse persistente, por exemplo: `Cat.hbm.xml`, `Dog.hbm.xml`, ou se estiver usando herança, `Animal.hbm.xml`.

5.1.3. class

Você pode declarar uma classe persistente utilizando o elemento `class`:

```
<class
        name="ClassName"                              (1)
        table="tableName"                             (2)
        discriminator-value="discriminator_value"     (3)
        mutable="true|false"                          (4)
        schema="owner"                                (5)
        catalog="catalog"                             (6)
        proxy="ProxyInterface"                        (7)
```

```
        dynamic-update="true|false"                    (8)
        dynamic-insert="true|false"                    (9)
        select-before-update="true|false"              (10)
        polymorphism="implicit|explicit"               (11)
        where="arbitrary sql where condition"          (12)
        persister="PersisterClass"                     (13)
        batch-size="N"                                 (14)
        optimistic-lock="none|version|dirty|all"       (15)
        lazy="true|false"                              (16)
        entity-name="EntityName"                       (17)
        check="arbitrary sql check condition"          (18)
        rowid="rowid"                                  (19)
        subselect="SQL expression"                     (20)
        abstract="true|false"                          (21)
        node="element-name"
/>
```

(1) `name` (opcional): O nome da classe Java inteiramente qualificado da classe persistente (ou interface); Se o atributo estiver ausente, assume-se que o mapeamento é para intidades não-POJO.

(2) `table` (opcional – valor default para nomes de classes não qualificadas) O nome da sua tabela do banco de dados.

(3) `discriminator-value` (opcional – valor default para o nome da classe): Um valor que distingue subclasses individuais, usadas para o comportamento polimorfico. Valores aceitos incluem `null` e `not null`

(4) `mutable` (opcional - valor default `true`): Especifica que instancias da classe são (ou não) mutáveis

(5) `schema` (opcional): Sobrepõe o nome do esquema especificado pelo elemento root `<hibernate-mapping>`.

(6) `catalog` (opcional): Sobrepõe o nome do catálogo especificado pelo elemento root `<hibernate-mapping>`.

(7) `proxy` (opcional): Especifica um interface para ser utilizada pelos proxies de inicialização tardia. Você pode especificar o nome da própria classe.

(8) `dynamic-update` (opcional, valor default `false`): Especifica que o SQL de `UPDATE` deve ser gerado em tempo de execução e conter apenas aquelas colunas cujos valores foram alterados.

(9) `dynamic-insert` (opcional, valor default `false`): Especifica que o SQL de `INSERT` deve ser gerado em tempo de execução e conter apenas aquelas colunas cujos valores não estão nulos.

(10) `select-before-update` (opcional, valor default `false`): Especifica que o Hibernate *never* deve executar um SQL de `UPDATE` a não ser que com certeza um objeto está atualmente modificado. Em certos casos (atualmente, apenas quando um objeto transiente foi associado com uma nova sessão utilizando `update()`), isto significa que o Hibernate ira executar uma instrução SQL de `SELECT` adicional para determinar se um `UPDATE` é necessário nesse momento.

(11) `polymorphism` (opcional, default para `implicit`): Determina se deve ser utilizado a query polimorfica implicita ou explicitamente.

(12) `where` (opicional) especifica um comando SQL `WHERE` arbitrário para ser usado quando da recuperação de objetos desta classe.

(13) `persister` (opcional): Espeicifca uma `ClassPersister` customizada.

(14) `batch-size` (opcional, valor default `1`) especifica um "tamanho de lote" para a recuperação de instancias desta classe pelo identificador.

(15) `optimistic-lock` (octional, valor default `version`): Determina a estratégia de bloqueio.

(16) `lazy` (opcional): A recuperação tardia pode ser completamente desabilitada, setando `lazy="false"`.

(17) `entity-name` (opcional, default para o nome da classe): O Hibernate3 permite uma classe ser mapeada multiplas vezes, (potencialmente,para diferentes tabelas), e permite mapeamentos de entidades que são representadas por Maps ou XML no nível Java. Nestes casos, você deve especificar um nome arbitrário explicitamente para a entidade. Veja Seção 4.4, “Modelos dinâmicos” e Capítulo 18, *Mapeamento XML* para maiores informações.

(18) `check` (opcional): Uma expressão SQL utilizada para gerar uma constraint de *verificação* de múltiplas linhas para a geração automática do esquema.

(19) `rowid` (opcional): O Hibernate poder usar as assim chamadas ROWIDs em bancos de dados que a suportam. Por exemplo, no Oracle, o Hibernate pode utilizar a coluna extra rowid para atualizações mais rápi-

das se você configurar esta opção para `rowid`. Um ROWID é uma implementação que representa de maneira detalhada a localização física de uma determinada tupla armazenado.

(20) `subselect` (opcional): Mapeia uma entidade imutavel e somente de leitura para um subconjunto do banco de dados. Útil se você quiser ter uma view em vez de uma tabela. Veja abaixo para mais informações.

(21) `abstract` (opcional): Utilizada para marcar superclasses abstratas em hierarquias `<union-subclass>`.

É perfeitamente aceitável para uma classe persitente nomeada ser uma interface. Você deverá então declarar as classes implementadas desta interface utilizando o elemento `<subclass>`. Você pode persistir qualquer classe de aninhada *estatica*. Você deverá especificar o nome da classe usando a forma padrão, por exemplo: `eg.Foo$Bar`.

Classes imutáveis, `mutable="false"`, não podem ser modificadas ou excluídas pela aplicação. Isso permite ao Hibernate fazer alguns aperfeiçoamentos de performance.

O atributo opcional `proxy` habilita a inicialização tardia das instâncias persistentes da classe. O Hibernate irá retornar CGLIB proxies como implementado na interface nomeada. O objeto persistente atual será carregado quando um método do proxy for invocado. Veja "Proxies para Inicialização Lazy" abaixo.

Polimorfismo *implícito* significa que instâncias de uma classe serão retornada por uma query que dá nome a qualquer superclasse ou interface implementada, ou a classe e as instancias de qualquer subclasse da classe será retornada por umq query que nomeia a classe por si. Polimorfismo *explícito* significa que instancias da classe serão retornadas apenas por queries que explicitamente nomeiam a classe e que queries que nomeiam as classes irão retornar apenas instancias de subclasses mapeadas dentro da declaração `<class>` como uma `<subclass>` ou `<joined-subclass>`. Para a maioria dos casos, o valor default `polymorphism="implicit"`, é apropriado. Polimorfismo explicito é útil quando duas classes distintas estão mapeadas para a mesma tabela (isso permite um classe "peso leve" que contem um subconjunto de colunas da tabela).

O atributo `persister` deixa você customizar a estratégia de persistência utilizada para a classe. Você pode, por exemplo, especificar sua prórpia subclasse do `org.hibernate.persister.EntityPersister` ou você pode criar uma implementação completamente nova da interface `org.hibernate.persister.ClassPersister` que implementa a persistência através de, por exemplo, chamadas a stored procedeures, serialização de arquivos flat ou LDAP. Veja `org.hibernate.test.CustomPersister` para um exemplo simples (de "persistencia" para uma `Hashtable`).

Observe que as configurações `dynamic-update` e `dynamic-insert` não sao herdadas pelas subclasses e assim podem tambem ser especificadas em elementos `<subclass>` ou `<joined-subclass>`. Estas configurações podem incrementar a performance em alguns casos, mas pode realmente diminuir a performance em outras. Use-as de forma bastante criteriosa.

O uso de `select-before-update` geralmente irá diminuir a performance. Ela é muito útil para prevenir que uma trigger de atualização no banco de dados seja ativada desnecessariamente, se você reconectar um nó de uma instancia desconectada em uma `Session`.

Se você ativar `dynamic-update`, você terá de escolher a estratégia de bloqueio otimista:

- `version` verifica a versão e a hora das colunas
- `all` cverifica todas as colunas
- `dirty` verifica as colunas modificadas, permitindo alguns updates concorrentes
- `none` não utiliza o bloqueio otimista

Nós *recomendamos* com muita enfase que você utilize a versão e a hora das colunas para o bloqueio otimista com o Hibernate. Esta é a melhor estratégia com respeito a performance e é a única estratégia que trata correta-

mente as modificações efetuadas em instancias desconectadas (por exemplo, quando `Session.merge()` é utilizado).

Não ha diferença entre uma view e uma tabela para o mapeamento do Hibernate, e como esperado isto é transparente no nível do banco de dados (observe que alguns bancos de dados não suportam views apropriadamente, especialmente com updates). Algumas vezes, você quer utilizar uma view, ma snão pode cria-la no banco de dados (por exemplo, com um esquema legado). Neste caso, você pode mapear uma entidade imutável e de somente leitura, para uma dada expressão SQL, que representa um subselect:

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
</class>
```

Declare as tabelas para sincronizar com esta entidade, garantindo que o auto-flush ocorra corretamente, e que as queries para esta entidade derivada não retornem dados desatualizados. O `<subselect>` está disponível tanto como um atributo como um elemento mapeado nested.

5.1.4. id

Classes mapeadas *precisam* declarar a coluna primary key da tabela do banco de dados. Muitas classes irão tambem ter uma propriedade ao estilo Java-Beans declarando o identificador unico de uma instancia. O elemento `<id>` define o mapeamento desta propriedade para a primary key.

```
<id
        name="propertyName"                                          (1)
        type="typename"                                              (2)
        column="column_name"                                         (3)
        unsaved-value="null|any|none|undefined|id_value"             (4)
        access="field|property|ClassName">                           (5)
        node="element-name|@attribute-name|element/@attribute|."

        <generator class="generatorClass"/>
</id>
```

(1) `name` (opcional): O nome do identificador.

(2) `type` (opcional): Um nome que indica o tipo no Hibernate.

(3) `column` (opcional – default para o a propridade name): O nome coluna primary key.

(4) `unsaved-value` (opcional - default para um valor "sensível"): Uma propriedade de identificação que indica que a instancia foi novamente instanciada (unsaved), diferenciando de instancias desconectadas que foram salvas ou carregadas em uma sessão anterior.

(5) `access` (opcional - valor default `property`): A estratégia que o Hiberante deve utilizar para acessar o valor da propriedade

Se o atributo `name` não for declarado, assume-se que a classe não tem a propriedade de identificação.

O atributo `unsaved-value` não é mais necessário no Hibernate 3.

Há declaração alternativa `<composite-id>` permite o acesso a dados legados com chaves compostas. Nós desencorajamos fortemente o seu uso por qualquer pessoa.

5.1.4.1. Generator

O elemento filho opcional `<generator>` nomeia uma classe Java usada para gerar identificadores unicos para instancias de uma classe persistente. Se algum parâmetro é requerido para configurar ou inicializar a instancia geradora, eles são passados utilizando o elemento `<param>`.

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

Todos os generators implementam a interface `org.hibernate.id.IdentifierGenerator`. Esta é uma interface bem simples; algumas aplicações podem prover sua própria implementação esepecializada. Entretanto, o Hibernate disponibiliza um conjunto de implementações internamente. Há nomes de atalhos para estes generators próprios:

`increment`
: gera identificadores dos tipos `long`, `short` ou `int` que são unicos apenas quando nenhum outro processo está inserindo dados na mesma tabela. *Não utilize em ambientes de cluster*.

`identity`
: suporta colunas de identidade em DB2, MySQL, MS SQL Server, Sybase e HypersonicSQL. O identificador retornado é do tipo `long`, `short` ou `int`.

`sequence`
: utiliza uma sequence em DB2, PostgreSQL, Oracle, SAP DB, McKoi ou um generator no Interbase. O identificador de retorno é do tipo `long`, `short` ou `int`.

`hilo`
: utiliza um algoritmo hi/lo para gerar de forma eficiente identificadores do tipo `long`, `short` ou `int`, a partir de uma tabela e coluna fornecida (por default `hibernate_unique_key` e `next_hi`) como fonte para os valores hi. O algoritmo hi/lo gera identificadores que são únicos apenas para um banco de dados particular.

`seqhilo`
: utiliza um algoritmo hi/lo para gerar de forma eficinete identificadores do tipo `long`, `short` ou `int`, a partir de uma sequence de banco de dados fornecida.

`uuid`
: utiliza um algortimo UUID de 128-bits para gerar identificadores do tipo string, unicos em uma rede(o endereço IP é utilizado). O UUID é codificado como um string de digitos hexadecimais de tamanho 32.

`guid`
: utiliza um string GUID gerado pelo banco de dados no MS SQL Server e MySQL.

`native`
: seleciona entre `identity`, `sequence` ou `hilo` dependendo das capacidades do banco de dados utilizado.

`assigned`
: deixa a aplicação definir um identificador para o objeto antes que o `save()` seja chamado. Esta é a estratégia default se nenhum elemento `<generator>` é especificado.

`select`
: retorna a primary key recuperada por uma trigger do banco de dados, selecionado uma linha pela chave

única e recuperando o valor da primary key.

`foreign`
: utiliza o identificador de um outro objeto associado. Normalmente utilizado em conjunto com uma associação primary key do tipo `<one-to-one>`.

`sequence-identity`
: uma estratégia de geração de sequence especializada que utiliza uma sequence de banco de dados para a geração de novos valores, combina com getGeneratedKeys de JDBC3 para devolver o valor do identificador como parte da execução de declaração de insert. Esta estratégia só é suportada por drives do Oracle 10g especificas para JDK 1.4. Comentários nestas declarações de inserts são nválido devido a um bug nos drivers de Oracle.

5.1.4.2. Algoritmo Hi/lo

Os geradores `hilo` e `seqhilo` fornecem duas implementações alternativas do algoritmo hi/lo, uma solução preferencial para a geração de identificadores. A primeira implementação requer uma tabela especial do banco de dados para manter o proximo valor "hi" disponível. A segunda utiliza uma seqüência do estilo Oracle (quando suportado).

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

Infelizemente, você não pode utilizar `hilo` quando estiver fornecendo sia propria `Connection` para o Hibernate. Quando o Hibernate está usando um datasource do servidor de aplicações para obter conexões suportadas com JTA, você precisa configurar adequadamente o `hibernate.transaction.manager_lookup_class`.

5.1.4.3. UUID algorithm

O UUID contem: o endereço IP, hora de inicio da JVM (com precisão de um quarto de segundo), a hora do sistema e um valor contador (unico dentro da JVM). Não é possivel obter o endereço MAC ou um endereço de memória do código Java, assim este é o melhor que pode ser feito sem utilizar JNI.

5.1.4.4. Colunas de identidade e sequencias

Para bancos de dados que suportam colunas de identidade (DB2, MySQL, Sybase, MS SQL), você pode utilizar uma geração de chave `identity`. Para bancos de dados que suportam sequencias (DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB) você pode utilizar a geração de chaves no estilo `sequence`. As duas estratégias requerem duas consultas SQL para inserir um novo objeto.

```
<id name="id" type="long" column="person_id">
        <generator class="sequence">
                <param name="sequence">person_id_sequence</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="person_id" unsaved-value="0">
        <generator class="identity"/>
</id>
```

Para desenvolvimento multi-plataforma, a estratégia `native` irá escolher entre as estratégias i `identity`, `sequence` e `hilo`, dependendo das capacidades do banco de dados utilizado.

5.1.4.5. Identificadores especificados

Se você quer que a aplicação especifique os identificadores (em vez do Hibernate gerá-los) você deve utilizar o gerador `assigned`. Este gerador especial irá utilizar o valor do identificador especificado para a propriedade de identificação do objeto. Este gerador é usado quando a primary key é a chave natural em vez de uma surrogate key. Este é o comportamento padrão se você não especificar um elemento `<generator>`.

Escolher o gerador `assigned` faz com que o Hibernate utilize `unsaved-value="undefined"`, forçando o Hibernate ir até o banco de dados para determinar se uma instância está transiente ou desasociada, a menos que haja uma versão ou uma propriedade timestamp, ou você pode definir `Interceptor.isUnsaved()`.

5.1.4.6. Primary keys geradas por triggers

Apenas para sistemas legados (o Hibernate nao gera DDL com triggers).

```
<id name="id" type="long" column="person_id">
        <generator class="select">
                <param name="key">socialSecurityNumber</param>
        </generator>
</id>
```

No exemplo acima, há uma única propriedade com valor nomeada `socialSecurityNumber` definida pela classe, uma chave natural, e uma surrogate key nomeada `person_id` cujo valor é gerado pro uma trigger.

5.1.5. composite-id

```
<composite-id
        name="propertyName"
        class="ClassName"
        mapped="true|false"
        access="field|property|ClassName">
        node="element-name|."

        <key-property name="propertyName" type="typename" column="column_name"/>
        <key-many-to-one name="propertyName class="ClassName" column="column_name"/>
        ......
</composite-id>
```

Para tabelas com uma chave composta, você pode mapear múltiplas propriedades da classe como propriedades de identificação. O elemento `<composite-id>` aceita o mapeamento da propriedade `<key-property>` e mapeamentos `<key-many-to-one>`como elements filhos.

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

Sua classe persistente *precisa* sobre escrever `equals()` e `hashCode()` para implementar identificadores compostos igualmente. E precisa também implementar `Serializable`.

Infelizmente, esta solução para um identificador composto significa que um objeto persistente é seu próprio identificador. Não há outro "handle" que o próprio objeto. Você mesmo precisa instanciar uma instância de outra classe persistente e preencher suas propriedades de identificação antes que você possa dar um `load()` para o estado persistente associado com uma chave composta. Nos chamamos esta solução de identificador composto *embedded* e não aconselhamos para aplicações sérias.

Uma segunda solução é o que podemos chamar de identificador composto *mapped* quando a propriedades de identificação nomeadas dentro do elemento `<composite-id>` estão duplicadas tando na classe persistente como em uma classe de identificação separada.

```
<composite-id class="MedicareId" mapped="true">
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

No exemplo, ambas as classes de identificação compostas, `MedicareId`, e a própria classe entidade tem propriedades nomeadas `medicareNumber` e `dependent`. A classe identificadora precisa sobrepor `equals()` e `hashCode()` e implementar `Serializable`. A desvantagem desta solução é obvia – duplicação de código.

Os seguintes atributos ão utilizados para especificar o mapeamento de um identificador composto:

- `mapped` mapped (opcional, valor default `false`): indica que um identificar composto mapeado é usado, e que as propriedades de mapeamento contidas refere-se tanto a classe entidade e a classe de identificação composta.
- `class` (opcional, mas requerida para um identificar composto mapeado): A classe usada como um identificador composto.

Nós iremos descrever uma terceira e as vezes mais conveniente solução, onde o identificador composto é implementado como uma classe componente na Seção 8.4, “. Componentes como identificadores compostos”. Os atributos descritos abaixo aplicam-se apenas para esta solução:

- `name` (opcional, requerida para esta solução): Uma propriedade do tipo componente que armazena o identificador composto (veja capítulo 9)
- `access` (opcional - valor default `property`): A estartégia Hibernate que deve ser utilizada para acessar o valor da propriedade.
- `class` (opcional - valor default para o tipo de propriedade determiando por reflexão) : A classe componente utilizada como um identificador composto (veja a próxima sessão).

Esta terceira solução, um *componente de identificação*, é o que nós recomendamos para a maioria das aplicações.

5.1.6. discriminator

O elemento `<discriminator>` é necessário para persistência polimórfica utilizando a estratégia de mapeamento table-per-class-hierarchy e declara uma coluna discriminadora da tabela. A coluna discriminadora contem valores de marcação que dizem a camada de persistência qual subclasse instanciar para uma linha particular. Um restrito conjunto de tipos que podem ser utilizados: `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

```
<discriminator
        column="discriminator_column"                   (1)
        type="discriminator_type"                       (2)
        force="true|false"                              (3)
        insert="true|false"                             (4)
        formula="arbitrary sql expression"              (5)
/>
```

(1) `column` (opcional - valor default `class`) o nome da coluna discriminadora
(2) `type` (opcional - valor default `string`) o nome que indica o tipo Hibernate
(3) `force` (opcional - valor default `false`) "força" o Hibernate a especificar valores discriminadores permitidos mesmo quando recuperando todas as instancias da classe root.
(4) `insert` (opcional - valor default para `true`) sete isto para `false` se sua coluna discriminadora é também parte do identificador composto mapeado. (Diz ao Hibernate para não incluir a coluna em comandos SQL `INSERT`s).
(5) `formula` (opcional) uma expressão SQL arbitraria que é e xecutada quando um tipo tem que ser avaliado. Permite discriminação baseada em conteúdo.

Valores atuais de uma coluna discriminada são especificados pelo atributo `discriminator-value` da `<class>` e elementos da `<subclass>`.

O atributo `force` é util (apenas) em tabelas contendo linhas com valores discriminadores "extras" que não estão mapeados para uma classe persistente. Este não é geralmente o caso.

Usando o atributo `formula` você pode declarar uma expressão SQL arbitrária que sera utilizada para avaliar o tipo de uma linha :

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
    type="integer"/>
```

5.1.7. version (opcional)

O elemento `<version>` é opcional e indica que a tabela possui dados versionados. Isto é particularmente útil se você planeja utilizar *transações longas* (veja abaixo):

```
<version
        column="version_column"                                     (1)
        name="propertyName"                                         (2)
        type="typename"                                             (3)
        access="field|property|ClassName"                           (4)
        unsaved-value="null|negative|undefined"                     (5)
        generated="never|always"                                    (6)
        insert="true|false"                                         (7)
        node="element-name|@attribute-name|element/@attribute|."
/>
```

(1) `column` (opcional - default a a propriedade name): O nome da coluna mantendo o numero da versão
(2) `name`: O nome da propriedade da classe persistente.
(3) `type` (opcional - valor default para `integer`): O tipo do numero da versão
(4) `access` (opcional - valor default `property`): A estratégia Hibernate que deve ser usada para acessar o valor da propriedade.
(5) `unsaved-value` (opcional – valor default para `undefined`): Um valor para a propriedade versão que indica que uma instancia é uma nova instanciada (unsaved), distinguindo de instancias desconectadas que foram salvas ou carregadas em sessões anteriores. ((`undefined` especifica que o valor da propriedade de identificação deve ser utilizado).
(6) `generated` (opcional - valor default `never`): Especifica que valor para a propriedade versão é na verdade gerado pelo banco de dados. Veja a discussão da Seção Seção 5.6, “Propriedades geradas”.
(7) `insert` (opcional - valor default para `true`): Especifica se a coluna de versão deve ser incluída no comando SQL de insert. Pode ser configurado como `false` se a coluna do banco de dados está definida com um valor default de `0`.

Números de versão podem ser dos tipos Hibernate `long`, `integer`, `short`, `timestamp` ou `calendar`.

A versão de uma propriedade timestamp nunca deve ser nula para uma instancia desconectada, assim o Hibernate irá identificar qualquer instância com uma versão nula ou timestamp como transiente, não importando qual estratégia para foi especificada para `unsaved-value`. *Declarando uma versão nula ou a propriedade timestamp é um caminho fácil para tratar problemas com reconexões transitivas no Hibernate, especialmente úteis para pessoas utilizando identificadores assinaldados ou chaves compostas!*

5.1.8. timestamp (opcional)

O elemento opcional `<timestamp>` indica que uma tabela contém dados timestamped. Isso tem por objetivo dar uma alternativa para versionamento. Timestamps são por natureza uma implementação menos segura do locking otimista. Entretanto, algumas vezes a aplicação pode usar timestamps em outros caminhos.

```
<timestamp
        column="timestamp_column"                                   (1)
        name="propertyName"                                         (2)
        access="field|property|ClassName"                           (3)
        unsaved-value="null|undefined"                              (4)
        source="vm|db"                                              (5)
        generated="never|always"                                    (6)
        node="element-name|@attribute-name|element/@attribute|."
/>
```

(1) `column` (opcional - valor default para a propriedade name): O nome da coluna que mantem o timestamp.

(2) `name`: O nome da propriedade no estilo JavaBeans do tipo `Date` ou `Timestamp` da classe persistente Java.

(3) `access` (opcional - valor default para `property`): A estretagia Hibernate que deve ser utilizada para acessar o valor da propriedade.

(4) `unsaved-value` (opcional - valor default `null`): Uma propriedade para a versão de que indica que uma instância é uma nova instanciada (unsaved), distinguindo-a de instancias desconectadas que foram salvas ou carregadas em sessões previas. (`undefined` especifica que um valor de propriedade de identificação deve ser utilizado)

(5) `source` (opcional - valor default para `vm`): De onde o Hibernate deve recuperar o valor timestamp? Do banco de dados ou da JVM corrente? Timestamps baseados em banco de dados levam a um overhead porque o Hibernate precisa acessar o banco de dados para determinar o "próximo valor", mas é mais seguro para uso em ambientes de "cluster". Observe também, que nem todos `Dialects` suportam a recuperação do timestamp corrente do banco de dados, enquando outros podem não ser seguros para utilização em bloqueios pela falta de precisão (Oracle 8 por exemplo)

(6) `generated` (opcional - valor default `never`): Especifica que o valor da propriedade timestamp é gerado pelo banco de dados. Veja a discussão Seção 5.6, “Propriedades geradas”.

Observe que `<timestamp>` é equivalente a `<version type="timestamp">`. E `<timestamp source="db">` é equivalente a `<version type="dbtimestamp">`.

5.1.9. property

O elemento `<property>` declara uma propriedade persistente de uma classe, no estilo JavaBean.

```
<property
        name="propertyName"                                         (1)
        column="column_name"                                        (2)
        type="typename"                                             (3)
        update="true|false"                                         (4)
        insert="true|false"                                         (4)
        formula="arbitrary SQL expression"                          (5)
        access="field|property|ClassName"                           (6)
        lazy="true|false"                                           (7)
        unique="true|false"                                         (8)
        not-null="true|false"                                       (9)
```

```
        optimistic-lock="true|false"                                    (10)
        generated="never|insert|always"                                 (11)
        node="element-name|@attribute-name|element/@attribute|."
        index="index_name"
        unique_key="unique_key_id"
        length="L"
        precision="P"
        scale="S"
/>
```

(1) `name`: o nome da propriedade, iniciando com letra minúscula.

(2) `column` (opcional - default para a propriedade name): o nome da coluna mapeada do banco de dados, Isto pode também ser especificado pelo(s) elemento(s) `<column>` aninhados.

(3) `type` (opcional): um nome que indica o tipo Hibernate.

(4) `update, insert` (opcional - valor default `true`): especifica que as colunas mapeadas devem ser incluidas nas instruções SQL de `UPDATE` e/ou `INSERT` . Setar ambas para to `false` permite uma propridade "derivada" pura cujo valor é inicializado de outra propriedade que mapeie a mesma coluna(s) ou por uma trigger ou outra aplicação.

(5) `formula` (opcional): uma expressão SQL que definie o valor para uma propriedade *calculada*. Propriedades calculadas nao tem uma coluna de mapeamento para elas.

(6) `access` (opcional – valor default `property`): A estratégia que o Hibernate deve utilizar para acessar o valor da propriedade

(7) `lazy` (opcional - valor default para `false`): Especifica que esta propriedade deve ser trazida de forma "lazy" quando a instancia da variável é acessada pela primeira vez (requer instrumentação bytecode em tempo de criação).

(8) `unique` (opcional): Habilita a geração de DDL de uma unica constraint para as colunas. Assim, permite que isto seja o alvo de uma `property-ref`.

(9) `not-null` (opcional): Habilita a geração de DDL de uma constraint de nulidade para as colunas.

(10) `optimistic-lock` (opcional - valor default `true`): Especifica se mudanças para esta propriedade requerem ou não bloqueio otimista. Em outras palavras, determina se um incremento de versão deve ocorrer quando esta propriedade está suja.

(11) `generated` (opcional - valor default `never`): Especifica que o valor da propriedade é na verdade gerado pelo banco de dados. Veja a discussão da seção Seção 5.6, “Propriedades geradas”.

typename pode ser:

1. O nome do tipo basico do Hibernate (ex., `integer, string, character, date, timestamp, float, binary, serializable, object, blob`).
2. O nome da classe Java com um tipo básico default (ex. `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob`).
3. O nome da classe Java serializable
4. O nome da classe de um tipo customizado (ex. `com.illflow.type.MyCustomType`).

Se você não especificar um tipo, o Hibernate ira utilizar reflexão sobre a propriedade nomeada para ter uma idéia do tipo Hibernate correto. O Hibernate ira tentar interpretar o nome da classe retornada, usando as regras 2, 3 e 4 nesta ordem. Entretanto, isto não é sempre suficiente Em certos casos, você ainda irá necessitar do atributo `type`. (Por exemplo, para distinguir entre `Hibernate.DATE` ou `Hibernate.TIMESTAMP`, ou para espcificar uma tipo ciustomizado.)

O atributo `access` permite você controlar como o Hibernate irá acessar a propriedade em tempo de execução. Por default, o Hibernate irá chamar os métodos get/set das propriedades. Se você especificar `access="field"`, o Hibernate ira bipassar os metodos get/set, acessnado o campo diretamente, usando reflexão. Voc epode especificar sua própria estratégia para acesso da propriedade criando uma classe que implemente a interface `org.hibernate.property.PropertyAccessor`.

Um recurso especialmente poderoso é o de propriedades derivadas. Estas propriedades são por definição read-only, e o valor da propriedade é calculado em tempo de execução. Você declara este calculo como uma expressão SQL, que traduz para clausula `SELECT` de uma subquery daquery SQL que carrega a instancia:

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
                WHERE li.productId = p.productId
                AND li.customerId = customerId
                AND li.orderNumber = orderNumber )"/>
```

Observe que você pode referenciar as entidades da própria tabela, através da não declaração de um alias para uma coluna particular (`customerId` no exemplo dado). Observe tambem que você pode usar o mapeamento de elemento aninhado `<formula>`, se você não gostar de usar o atributo.

5.1.10. many-to-one

Uma associação ordinária para outra classe persistente é declarada usando o elemento `many-to-one`. O modelo relacional é uma associação many-to-one: a uma foreign key de uma tabela referenciando a primary key da tabela destino.

```
<many-to-one
        name="propertyName"                                         (1)
        column="column_name"                                        (2)
        class="ClassName"                                           (3)
        cascade="cascade_style"                                     (4)
        fetch="join|select"                                         (5)
        update="true|false"                                         (6)
        insert="true|false"                                         (6)
        property-ref="propertyNameFromAssociatedClass"              (7)
        access="field|property|ClassName"                           (8)
        unique="true|false"                                         (9)
        not-null="true|false"                                       (10)
        optimistic-lock="true|false"                                (11)
        lazy="proxy|no-proxy|false"                                 (12)
        not-found="ignore|exception"                                (13)
        entity-name="EntityName"                                    (14)
        formula="arbitrary SQL expression"                          (15)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"
/>
```

(1) `name`: O nome da propriedade.

(2) `column` (opcional): O nome da coluna foreign key. Isto pode também ser especificado através de elementos aninhados `<column>`.

(3) `class` (opcional – default para o tipo de propriedade determinado pela reflexão). O nome da classe associada.

(4) `cascade` (opcional): Especifica quais operações dever ser em cascata do objeto pai para o objeto associado.

(5) `fetch` (opcional - default para `select`): Escolhe entre recuperação outer-join ou recuperação seqüencial.

(6) `update, insert` (opcional - valor default `true`): especifica que as colunas mapeadas dever ser incluidas em instruções SQL de `UPDATE` e/ou `INSERT`. Setando ambas para `false` você permite uma associação "derivada" pura cujos valores são inicializados de algumas outras propriedades que mapeiam a mesma coluna ou por uma trigger ou outra aplicação.

(7) `property-ref`: (opcional) O nome da propriedade da classe associada que faz a junção desta foreign key. Se não especificada, a primary key da classe associada será utilizada.

(8) `access` (opcional - valor default `property`): A estrategia que o Hibernate deve utilizar para acessar o valor da propriedade.

(9) `unique` (opcional): Habilita a geração DDL de uma constraint unique para a coluna foreign key. Alem disso, permite ser o alvo de uma `property-ref`. Isso torna a associação multipla efetivamente um para um.

(10) `not-null` (opcional): Habilita a geração DDL de uma constraint de nulidade para as foreign keys.

(11) `optimistic-lock` (opcional - valor default `true`): Especifica se mudanças desta propriedade requerem ou não travamento otimista. Em outras palavras, determina se um incremento de versão deve ocorrer quando esta propriedade está suja.

(12) `lazy`(opcional – valor default `proxy`): Por default, associações de ponto unico são envoltas em um proxie. `lazy="no-proxy"` especifica que a propriedade deve ser trazida de forma tardia quando a instancia da variável é acessada pela primeira vez (requer instrumentação bytecode em tempo de criação) `lazy="false"` especifica que a associação será sempre recuperada fortemente.

(13) `not-found` (opcional - valor default `exception`): Especifica como as foreign keys que referenciam linhas ausentes serão tratadas: `ignore` irá tratar a linha ausente como uma associação de null

(14) `entity-name` (opcional): O nome da entidade da classe associada.

Setar o valor do atributo `cascade` para qualquer valor significativo diferente de `none` irá propagar certas operações ao objeto associado. Os valores significativos são os nomes das operações básicas do Hibernate, `persist, merge, delete, save-update, evict, replicate, lock, refresh`, assim como os valores especiais `delete-orphan` e `all` e combinações de nomes de operações separadas por vírgula, como por exemplo, `cascade="persist,merge,evict"` ou `cascade="all,delete-orphan"`. Veja a seção Seção 10.11, “Persistência transitiva” para uma explicação completa. Note que associações valoradas simples (associações muitos-pra-um, e um-pra-um) não suportam orphan delete.

Uma típica declaração `muitos-pra-um` se parece com esta:

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

O atributo `property-ref` deve apenas ser usado para mapear dados legados onde uma foreign key se referencia a uma chave exclusiva da tabela associada que não seja à primary key. Este é um modelo relacional desagradável. Por exemplo, suponha que a classe `Product` tenha um número seqüencial exclusivo, que não é a primary key. (O atributo `unique` controla a geração de DDL do Hibernate com a ferramenta SchemaExport.)

```
<property name="serialNumber" unique="true" type="string" column="SERIAL_NUMBER"/>
```

Então o mapeamento para `OrderItem` poderia usar:

```
<many-to-one name="product" property-ref="serialNumber" column="PRODUCT_SERIAL_NUMBER"/>
```

Porém, isto obviamente não é indicado, nunca.

Se a chave exclusiva referenciada engloba múltiplas propriedades da entidade associada, você deve mapear as propriedades referenciadas dentro de um elemento chamado `<properties>`

Se a chave exclusiva referenciada é a propriedade de um componente, você pode especificar um caminho para a propriedade.

```
<many-to-one name="owner" property-ref="identity.ssn" column="OWNER_SSN"/>
```

5.1.11. one-to-one (um-pra-um)

Uma associação um-pra-um para outra classe persistente é declarada usando um elemento `one-to-one` .

```
<one-to-one
```

```
        name="propertyName"                                          (1)
        class="ClassName"                                            (2)
        cascade="cascade_style"                                      (3)
        constrained="true|false"                                     (4)
        fetch="join|select"                                          (5)
        property-ref="propertyNameFromAssociatedClass"               (6)
        access="field|property|ClassName"                            (7)
        formula="any SQL expression"                                 (8)
        lazy="proxy|no-proxy|false"                                  (9)
        entity-name="EntityName"                                     (10)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        foreign-key="foreign_key_name"
/>
```

(1) `name`: O nome da propriedade.

(2) `class` (opcional – default para o tipo da propriedade definido via reflection): O nome da classe associada.

(3) `cascade` (opcional): Especifica qual operação deve ser cascateada do objeto pai para o objeto associado.

(4) `constrained` (opcional): Especifica que uma constraint foreign key na primary key da tabela mapeada referencia a tabela da classe associada, Esta opção afeta a ordem em queh `save()` e `delete()` são cascateadas, e determina se a associação pode ser substituída (isto também é usado pela ferramenta schema export).

(5) `fetch` ((opcional – valor default `select`): Escolhe entre outer-join fetching ou sequential select fetching.

(6) `property-ref`(opcional): O nome da propriedade da classe associada que é ligada a primary key desta classe. Se não for especificada, a primary key da classe associada é utilizada.

(7) `access` (opcional - valor default padrão `property`): A estratégia que o Hibernate pode usar para acessar o valor da propriedade.

(8) `formula` (opcional): Quase todas as associações um-pra-um mapeiam para a primary key da entidade forte. Em uma caso raro, que não é o nosso caso, você pode especificar uma outra coluna, colunas ou expressões utilizando uma formula SQL. (Veja `org.hibernate.test.onetooneformula` para um exemplo).

(9) `lazy` (opcional – valor default `proxy`): Por default, associações single point são proxied. `lazy="no-proxy"` especifica que a propriedade deve ser fetched lazily quando o atributo é acessado pela primeira vez (requer build-time bytecode instrumentation). `lazy="false"` especifica que a associação vai sempre ser avidamente fetched. *Note que se `constrained="false"`, proxing é impossível e o Hibernate vai ávido fetch a associação!*

(10) `entity-name` (opcional): O nome da entidade da classe associada.

Existem duas variedades de associações um-pra-um:

- associações primary key
- associações de foreign key exclusiva

Associações primary key não necessitam de uma coluna extra de tabela; se duas linhas são relacionadas pela associação então as duas linhas da tabela dividem a mesmo valor da primary key. Assim, se você quer que dois objetos sejam relacionados por uma associação primary key, você deve ter certeza que eles são assinados com o mesmo valor identificador!

Para uma associação primary key, adicione os seguintes mapeamentos em `Employee` e `Person`, respectivamente.

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

Agora nós devemos assegurar que as primary keys da linhas relacionadas nas tabelas PERSON e EMPLOYEE são iguais. Nós usamos uma estratégia especial de geração de identificador do Hibernate chamada `foreign`:

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

Uma nova instância de `Person` salva recentemente é então assinada com o mesmo valor da primary key da instância de `employee` referenciada com a propriedade `employee` daquela `Person`.

Alternativamente, uma foreign key com uma constraint unique , de `Employee` para `Person`, pode ser expressa como:

```
<many-to-one name="person" class="Person" column="PERSON_ID" unique="true"/>
```

E esta associação pode ser feita de forma bi-direcional adicionando o seguinte no mapeamento de `Person`:

```
<one-to-one name"employee" class="Employee" property-ref="person"/>
```

5.1.12. natural-id

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

Embora nós recomendemos o uso de surrogate keys como primary keys, você deve ainda identificar chaves naturais para todas as entidades. Uma chave natural é uma propriedade ou combinação de propriedades que é exclusiva e não nula. Se não pude ser modificada, melhor ainda. Mapeie as propriedades da chave natural dentro do elemento `<natural-id>`. O Hibernate irá gerar a chave exclusiva necessária e as constraints de nullability , e seu mapeamento será apropriadamente auto documentado.

Nós recomendamos com enfase que você implemente `equals()` e `hashCode()` para comparar as propriedades da chave natural da entidade.

Este mapeamento não tem o objetivo de uso com entidades com primary keys naturais.

- `mutable` mutable (opcional, valor default`false`): Por default, propriedades naturais identificadoras são consideradas imutáveis (constante).

5.1.13. componente, componente dinâmico.

O elemento`<component>` mapeia propriedades de um objeto filho para colunas da tabela de uma classe pai. Componentes podem, um após o outro, declarar suas próprias propriedades, componentes ou coleções. Veja "Components" abaixo.

```
<component
        name="propertyName"                (1)
        class="className"                  (2)
        insert="true|false"                (3)
        update="true|false"                (4)
```

```
        access="field|property|ClassName"   (5)
        lazy="true|false"                   (6)
        optimistic-lock="true|false"        (7)
        unique="true|false"                 (8)
        node="element-name|."
>

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

(1) `name`: O nome da propriedade.
(2) `class` (opcional – valor default para o tipo de propriedade determinada por reflection): O nome da classe (filha) do componente.
(3) `insert`: As colunas mapeadas aparecem nos SQL de `INSERT`s?
(4) `update`: As colunas mapeadas aparecem nos SQL de `UPDATE`s?
(5) `access` (opcional – valor default `property`): A estratégia que o Hibernate pode usar para acessar o valor da propriedade.
(6) `lazy` (opcional - valor default `false`): Especifica que este componente deve ser fetched lazily quando o atributo for acessado pela primeira vez (requer build-time bytecode instrumentation).
(7) `optimistic-lock` (opcional – valor default `true`): Especifica que atualizações para este componente requerem ou não aquisição de um lock otimista. Em outras palavras, determina se uma versão de incremento deve ocorrer quando esta propriedade estiver modificada.
(8) `unique` (opcional – valor default `false`): Especifica que existe uma unique constraint em todas as colunas mapeadas do componente.

A tag filha `<property>` acrescenta a propriedade de mapeamento da classe filha para colunas de uma tabela.

O elemento `<component>` permite um sub-elemento `<parent>` mapeie uma propriedade da classe do componente como uma referencia de volta para a entidade que o contém.

O elemento `<dynamic-component>` permite que um `Map` possa ser mapeado como um componente onde os nomes das propriedades referem-se para as chaves no mapa, veja Seção 8.5, “Componentes Dinâmicos”.

5.1.14. propriedades

O elemento `<properties>` permite a definição de um grupo com nome, lógico de propriedades de uma classe. O uso mais importante do construtor é que este permite uma combinação de propriedades para ser o objetivo de uma `property-ref`. É também um modo conveninente para definir uma unique constraint de múltiplas colunas.

```
<properties
        name="logicalName"                  (1)
        insert="true|false"                 (2)
        update="true|false"                 (3)
        optimistic-lock="true|false"        (4)
        unique="true|false"                 (5)
>

        <property ...../>
        <many-to-one .... />
        ........
</properties>
```

(1) `name`:: O nome lógico do agrupamento – *não* é o nome atual de propriedade.
(2) `insert`: As colunas mapeadas aparecem nos SQL de `INSERT`s?
(3) `update`: As colunas mapeadas aparecem nos SQL de `UPDATE`s?
(4) `optimistic-lock` (opcional – valor default `true`): Especifica que atualizações para estes componentes re-

querem ou não aquisição de um lock otimista. Em outras palavras, determina se uma versão de incremento deve ocorrer quando estas propriedades estiverem modificadas.

(5) `unique` (opcional – valor default `false`): Especifica que uma unique constraint existe em todas as colunas mapeadas do componente.

Por exemplo, se nós temos o seguinte mapeamento de `<properties>`:

```
<class name="Person">
    <id name="personNumber"/>
    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

Então nós podemos ter uma associação de dados herdados que referem a esta chave exclusiva da tabela `Person`, ao invés de se referirem a chave primária:

```
<many-to-one name="person"
         class="Person" property-ref="name">
    <column name="firstName"/>
    <column name="initial"/>
    <column name="lastName"/>
</many-to-one>
```

Nós não recomendamos o uso deste tipo de coisa fora do contexto de mapeamento de dados herdados.

5.1.15. subclass (subclasse)

Finalmente, a persistência polimórfica requer a declaração de cada subclasse da classe de persistência raiz. Para a estratégia de mapeamento table-per-class-hierarchy, a declaração `<subclass>` deve ser usada.

```
<subclass
        name="ClassName"                                (1)
        discriminator-value="discriminator_value"       (2)
        proxy="ProxyInterface"                          (3)
        lazy="true|false"                               (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">

        <property .... />
        .....
</subclass>
```

(1) `name`: O nome de classe completamente qualificada da subclasse.

(2) `discriminator-value` (opcional – valor default o nome da classe): Um valor que distingue subclasses individuais.

(3) `proxy` (opcional): Especifica a classe ou interface que usará os proxies de inicialização atrasada.

(4) `lazy` (opcional, valor default `true`): Configurar `lazy="false"` desabilitará o uso de inicialização atrasada.

Cada subclasse deve declarar suas próprias propriedades persistentes e subclasses. As propriedades `<version>` e `<id>` são configuradas para serem herdades da classe raiz. Cada subclasse numa hierarquia deve definir um único `discriminator-value`. Se nenhum for especificado, o nome da classe Java completamente qualificada será usada.

Para informações sobre mapeamento de heranças, veja o Capítulo 9, *Mapeamento de Herança*.

5.1.16. joined-subclass

Alternativamente, cada subclasse pode ser mapeada para sua própria tabela (Estratégia de mapeamento table-per-subclass). O estado herdado é devolvido por associação com a tabela da superclasse. Nós usamos o elemento `<joined-subclass>`.

```
<joined-subclass
        name="ClassName"                    (1)
        table="tablename"                   (2)
        proxy="ProxyInterface"              (3)
        lazy="true|false"                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

(1) `name`: O nome da classe completamente qualificada da subclasse.
(2) `table`: O nome da tabela da subclasse.
(3) `proxy` (opcional): Especifica a classe ou interface para usar os proxies de recuperação atrasada.
(4) `lazy` (opcional, valor default `true`): Fixanr `lazy="false"` desabilita o uso recuperação atrasada.

A coluna discriminator requerida para esta estratégia de mapeamento. Porém, cada subclasse deve declarar uma coluna de tabela com o identificador do objeto usando o elemento `<key>`. O mapeamento no início do capítulo poderia ser re-escrito assim:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
                    <key column="CAT"/>
                    <property name="name" type="string"/>
                </joined-subclass>
        </class>
```

```
        <class name="eg.Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

Para informações de mapeamentos de herança, veja Capítulo 9, *Mapeamento de Herança*.

5.1.17. union-subclass

Uma terceira opção é mapear para tabelas apenas as classes concretas de uma hierarquia de heranças, (a estratégia table-per-concrete-class) onde cada tabela define todos os estados persistentes da classe, incluindo estados herdados. No Hibernate, não é absolutamente necessário mapear explicitamente como hierarquia de heranças. Você pode simplesmente mapear cada classe com uma declaração `<class>` separada. Porém, se você deseja usar associações polimórficas (por exemplo: uma associação para a superclasse de sua hierarquia), você precisa usar o mapeamento `<union-subclass>`.

```
<union-subclass
        name="ClassName"                    (1)
        table="tablename"                   (2)
        proxy="ProxyInterface"              (3)
        lazy="true|false"                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <property .... />
        .....
</union-subclass>
```

(1) `name`: O nome da subclasse completamente qualificada.
(2) `table`: O nome da tabela da subclasse.
(3) `proxy` (opcional): Especifica a classe ou interface para usar os proxies de recuperação atrasada.
(4) `lazy` (opcional, defaults to `true`): Setting `lazy="false"` disables the use of lazy fetching. `lazy` (opcional, valor default `ptrue`): Fixando `lazy="false"` desabilita o uso da recuperação atrasada.

A coluna discriminatõria não é requerida para esta estratégia de mapeamento.

Para informações sobre mapeamentos de herança, veja Capítulo 9, *Mapeamento de Herança*.

5.1.18. join

Usando o elemento `<join>>`, é possível mapear propriedades de uma classe para várias tabelas, desde que haja um relacionamento um-para-um entre as tabelas.

```
<join
        table="tablename"                       (1)
        schema="owner"                          (2)
        catalog="catalog"                       (3)
        fetch="join|select"                     (4)
        inverse="true|false"                    (5)
        opcional="true|false">                  (6)
```

```
        <key ... />

        <property ... />
        ...
</join>
```

(1) `table`: O nome da tabela associada.
(2) `schema` (opcional): Sobrepõe o nome do esquema especificado pelo elemento raiz `<hibernate-mapping>`.
(3) `catalog` (opcional): Sobrepõe o nome do catálogo especificado pelo elemento raiz`<hibernate-mapping>`.
(4) `fetch`(opcional – valor default `join`): Se setado para `join`, o padrão, o Hibernate irá usar um inner join para restaurar um `join` definido por uma classe ou suas subclasses e uma outer join para um `join` definido por uma subclasse. Se setado para `select`, então o Hibernate irá usar uma seleção seqüencial para um `<join>` definida numa subclasse, que irá ser emitido apenas se uma linha se concentrar para representar uma instância da subclasse. Inner joins irá ainda ser usado para restaurar um `<join>` definido pela classe e suas superclasses.
(5) `inverse` (opcional – valor default `false`): Se habilitado, o Hibernate não irá tentar inserir ou atualizar as propriedades definidas por este join.
(6) `opcional` (opcional – valor default `false`): Se habilitado, o Hibernate irá inserir uma linha apenas se as propriedades definidas por esta junção não forem nulas e irá sempre usar uma outer join para recuperar as propriedades.

Por exemplo, a informação de endereço para uma pessoa pode ser mapeada para uma tabela separada (enquanto preservando o valor da semântica de tipos para todas as propriedades):

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

Esta característica é útil apenas para modelos de dados legados, nós recomendamos menos tabelas do que classes e um modelo de domínio bem granulado. Porém, é útil para ficar trocando entre estratégias de mapeamento de herança numa hierarquia simples, como explicado mais a frente.

5.1.19. key

Nós vimos que o elemento `<key>` surgiu algumas vezes até agora. Ele aparece em qualquer lugar que o elemento pai define uma junção para a nova tabela, e define a foreign key para a tabela associada, que referencia a primary key da tabela original.

```
<key
        column="columnname"                     (1)
        on-delete="noaction|cascade"            (2)
        property-ref="propertyName"             (3)
        not-null="true|false"                   (4)
        update="true|false"                     (5)
        unique="true|false"                     (6)
/>
```

(1) . `column` (opcional): O nome da coluna foreign key. Isto também pode ser especificado por aninhamento de elemento(s) `<column>`.

(2) `on-delete` (opcional, valor default `noaction`): Especifica se a constraint da foreign key no banco de dados esta habilitada para cascade delete .

(3) `property-ref` (opcional): Especifica que a foreign key se refere a colunas que não são primary key da tabela original. (Util para base de dados legadas.)

(4) `not-null` (opcional): Especifica que a coluna foreign key não aceita valores nulos (isto é implícito em qualquer momento que a foreign key também fizer parte da primary key).

(5) `update` (opcional): Especifica que a foreign key nunca deve ser atualizada (isto é implícito em qualquer ponto onde a foreign key também fizer parte da primary key).

(6) `unique` (opcional): Especifica que a foreign key deve ter uma constraint unique (isso está implícito em qualquer momento que a foreign key também fizer parte da primary key).

Nós recomendamos que para sistemas que a performance de delete seja importante, todas as chaves deve ser definida `on-delete="cascade"`, e o Hibernate irá usar uma constraint a nível de banco de dados `ON CASCADE DELETE`, ao invés de muitas instruções `DELETE`. Esteja ciente que esta característica é um atalho da estratégia usual de optimistic locking do Hibernate para dados versionados.

Os atributos `not-null` e `update` são úteis quando estamos mapeamos uma associação unidirecional um para muitos. Se você mapear uma asociação unidirecional um-para-muitos para uma foreign key que não aceita valores nulos, você *deve* declarar a coluna chave usando `<key not-null="true">`.

5.1.20. elementos column e formula

Qualquer elemento de mapeamente que aceita um atributo `column` irá aceitar alternativamente um subelemento `<column>`. Da mesma forma, `formula` é uma alternativa para o atributo `formula`.

```
<column
        name="column_name"
        length="N"
        precision="N"
        scale="N"
        not-null="true|false"
        unique="true|false"
        unique-key="multicolumn_unique_key_name"
        index="index_name"
        sql-type="sql_type_name"
        check="SQL expression"
        default="SQL expression"/>
```

```
<formula>SQL expression</formula>
```

O atributo `column` e `formula` podem até ser combinados dentro da mesma propriedade ou associação mapeando para expressar, por exemplo, associações exóticas.

```
<many-to-one name="homeAddress" class="Address"
        insert="false" update="false">
    <column name="person_id" not-null="true" length="10"/>
    <formula>'MAILING'</formula>
</many-to-one>
```

5.1.21. import

Suponha que a sua aplicação tem duas classes persistentes com o mesmo nome, e você não quer especificar o nome qualificado (do pacote) nas queries do Hibernate. As Classes devem ser "importadas" explicitamente, de preferência contando com `auto-import="true"`. Você pode até importar classes e interfaces que não estão explicitamente mapeadas.

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"             (1)
        rename="ShortName"            (2)
/>
```

(1) `class`: O nome qualificado (do pacote) de qualquer classe Java.

(2) `rename` (opcional – valor default, o nome da classe não qualificada): Um nome que pode ser usado numa linguagem de consulta.

5.1.22. any

Existe mais um tipo de propriedade de mapeamento. O elemento de mapeamento `<any>` define uma associação polimórfica para classes de múltiplas tabelas. Este tipo de mapeamento sempre requer mais de uma coluna. A primeira coluna possui o tipo da entidade associada. A outra coluna que ficou possui o identificador. É impossível especificar uma restrição de foreign key para este tipo de associação, assim isto claramente não é visto como um caminho usual para associações (polimórficas) de mapeamento. Você deve usar este mapeamento apenas em casos muito especiais (exemplo: audit logs, dados de sessão do usuário, etc).

O atributo `meta-type` permite a aplicação especificar um tipo adaptado que mapeia valores de colunas de banco de dados para classes persistentes que tem propriedades identificadoras do tipo especificado através do `id-type`. Você deve especificar o mapeamento de valores do meta-type para nome de classes.

```
<any name="being" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="propertyName"                     (1)
        id-type="idtypename"                    (2)
        meta-type="metatypename"                (3)
        cascade="cascade_style"                 (4)
        access="field|property|ClassName"       (5)
        optimistic-lock="true|false"            (6)
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

(1) `name`: o nome da propriedade.

(2) `id-type`: o tipo identificador.

(3) `meta-type` (opcional – valor default `string`): Qualquer tipo que é permitido para um mapeamento discriminador.

(4) `cascade` (opcional – valor default `none`): o estilo do cascade.

(5) `access` (opcional – valor default `property`): A estratégia que o hibernate deve usar para acessar o valor da propriedade.

(6) `optimistic-lock` (opcional - valor default`true`): Especifica que as atualizações para esta propriedade requerem ou não aquisição da trava otimista. Em outras palavras, define se uma versão de incremento deve ocorrer se esta propriedade está modificada.

5.2. Tipos do Hibernate

5.2.1. Entidades e valores

Para entender o comportamento de vários objetos em nível de linguagem de Java a respeito do serviço de persistência, nós precisamos classificá-los em dois grupos.

Uma *entidade* existe independentemente de qualquer outro objeto guardando referências para a entidade. Em contraste com o modelo usual de Java que um objeto não referenciado é coletado pelo garbage collector. Entidades devem ser explicitamente salvas ou deletada (exceto em operações de salvamento ou deleção que possam ser executada em *cascata* de uma entidade pai para seus filhos). Isto é diferente do modelo ODMG de persistência do objeto por acessibilidade – e corresponde quase a como objetos de aplicações são geralmente usados em grandes sistemas. Entidades suportam referências circulares e comuns. Eles podem ser versionadas.

Uma entidade em estado persistente consiste de referências para outras entidades e instâncias de tipos de *valor*. Valores são primitivos, coleções (não o que tem dentro de uma coleção), componentes e certos objetos imutáveis. Entidades distintas, valores (em coleções e componentes particulares) *são* persistidos e apagados por acessibilidade. Visto que objetos value (e primitivos) são persistidos e apagados junto com as entidades que os contém e não podem ser versionados independentemente. Valores têm identidade não independente, assim eles não podem ser comuns para duas entidades ou coleções.

Até agora, nós estivemos usando o termo "classe persistente" para referir a entidades. Nós iremos continuar a fazer isto. Falando a rigor, porém, nem todas as classes definidas pelo usuário com estados persistentes são entidades. Um *componente* é uma classe de usuário definida com valores semânticos. Uma propriedade de Java de tipo `java.lang.String` também tem um valor semêntico. Dada esta definição, nós podemos dizer que todos os tipos (classes) fornecida pelo JDK tem tipo de valor semântico em Java, enquanto que tipos definidos pelo usuário pode ser mapeados com entidade ou valor de tipo semântico. Esta decisão pertence ao desenvolvedor da aplicação. Uma boa dica para uma classe entidade em um modelo de domínio são referências comuns para uma instância simples daquela classe, enquanto a composição ou agregação geralmente se traduz para um valor de tipo.

Nós iremos rever ambos os conceitos durante toda a documentação.

O desafio pe mapear o sistema de tipo de Java (e a definição do desenvolvedor de entidades e tipos de valor) para o sistema de tipo SQL/banco de dados. A ponte entre ambos os sistemas é fornecido pelo Hibernate: para entidades que usam `<class>`, `<subclass>` e assim por diante. Para tipos de valores nós usamos `<property>`, `<component>`, etc, geralmente com um atributo `type`. O valor deste atributo é o nome de um *tipo de mapeamento* do Hibernate. O Hibernate fornece muitos mapeamentos (para tipos de valores do JDK padrão) ut of the box. Você pode escrever os seus próprios tipos de mapeamentos e implementar sua estratégia de conversão adaptada, como você verá adiante.

Todos os tipos internos do hibernate exceto coleções suportam semânticas nulas.

5.2.2. Valores de tipos básicos

O tipos internos de mapeamentos básicos podem ser a grosso modo categorizado como:

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`
: Tipos de mapeamentos de classes primitivas ou wrapper Java especificos (vendor-specific) para tipos de coluna SQL. Boolean, `boolean`, `yes_no` são todas codificações alternativas para um `boolean` ou `java.lang.Boolean` do Java.

`string`
Um tipo de mapeamento de `java.lang.String` para `VARCHAR` (ou `VARCHAR2` no Oracle).

`date, time, timestamp`
Tipos de mapeamento de `java.util.Date` e suas subclasses para os tipos SQL `DATE`, `TIME` e `TIMESTAMP` (ou equivalente).

`calendar, calendar_date`
Tipo de mapeamento de `java.util.Calendar` para os tipos SQL `TIMESTAMP` e `DATE` (ou equivalente).

`big_decimal, big_integer`
Tipo de mapeamento de `java.math.BigDecimal` e `java.math.BigInteger` para `NUMERIC` (ou `NUMBER` no Oracle).

`locale, timezone, currency`
Tipos de mapeamentos de `java.util.Locale`, `java.util.TimeZone` e `java.util.Currency` para `VARCHAR` (ou `VARCHAR2` no Oracle). Instâncias de f `Locale` e `Currency` são mapeados para seus códigos ISO. Instâncias de `TimeZone` são mapeados para seu `ID`.

`class`
um tipo de mapeamento de `java.lang.Class` para `VARCHAR` (ou `VARCHAR2` no Oracle). Uma `Class` é mapeada pelo seu nome qualificado (completo).

`binary`
Mapeia arrays de bytes para um tipo binário de SQL apropriado.

`text`
Mapeia strings grandes de Java para um tipo SQL `CLOB` ou `TEXT`.

`serializable`
Mapeia tipos Java serializáveis para um tipo binário SQL apropriado. Você pode também indicar o tipo `serializable` do Hibernate com o nome da classe ou interface Java serializável que não é padrão para um tipo básico.

`clob, blob`
Tipos de mapeamentos para as classes JDBC `java.sql.Clob` e `java.sql.Blob`. Estes tipos podem ser inconveniente para algumas aplicações, visto que o objeto blob ou clob pode não ser reusado fora de uma transação. (Além disso, o suporte de driver é imcompleto e inconsistente.)

`imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date, imm_serializable, imm_binary`
Mapeando tipos para o que geralmente são consideradas tipos mutáveis de Java, onde o Hibernate faz determinadas otimizações apropriadas somente para tipos imutáveis de Java, e a aplicação trata o objeto como imutável. Por exemplo, você não deve chamar `Date.setTime()` para uma instância mapeada como `imm_timestamp`. Para mudar o valor da propriedade, e ter a mudança feita persistente, a aplicação deve atribuir um novo objeto (nonidentical) à propriedade.

Identificadores únicos das entidades e coleções podem ser de qualquer tipo básico exceto `binary`, `blob` ou `clob`. (Identificadores compostos também são permitidos, veja abaixo.)

Os tipos de valores básicos têm suas constantes `Type` correspondentes definidas em `org.hibernate.Hibernate`. Por exemplo, `Hibernate.STRING` representa o tipo `string`.

5.2.3. Tipos de valores personalizados

É relativamente fácil para desenvolvedores criar seus próprios tipos de valor. Por exemplo, você pode querer persistir propriedades do tipo `java.lang.BigInteger` para colunas `VARCHAR`. O Hibernate não fornece um tipo correspondente para isso. Mas os tipos adaptados não são limitados a mapeamento de uma propriedade (ou elemento de coleção) a uma única coluna da tabela. Assim, por exemplo, você pôde ter uma propriedade Java `getName()/setName()` do tipo `java.lang.String` que é persistido para colunas `FIRST_NAME`, `INITIAL`, `SURNAME`.

Para implementar um tipo personalizado, implemente `org.hibernate.UserType` ou `org.hibernate.CompositeUserType` e declare propriedades usando o nome qualificado da classe do tipo. Veja `org.hibernate.test.DoubleStringType` para ver o tipo das coisas que são possíveis.

```
<property name="twoStrings" type="org.hibernate.test.DoubleStringType">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

Observe o uso da tag `<column>` para mapear uma propriedade para colunas múltiplas.

As interfaces `CompositeUserType`, `EnhancedUserType`, `UserCollectionType`, e `UserVersionType` fornecem suporte para usos mais especializados.

Você pode mesmo fornecer parâmetros a um `UserType` no arquivo de mapeamento. Para isto, seu `UserType` deve implementar a interface `org.hibernate.usertype.ParameterizedType`. Para fornecer parâmetros a seu tipo personalizado, você pode usar o elemento `<type>` em seus arquivos de mapeamento.

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

O `UserType` pode agora recuperar o valor para o parâmetro chamado `default` da `Propriedade` do passado a ele.

Se você usar freqüentemente um determinado `UserType`, pode ser útil definir um nome mais curto para ele. Você pode fazer isto usando o elemento `<typedef>`. Typedefs atribui um nome a um tipo personalizado, e pode também conter uma lista de valores default de parâmetro se o tipo for parametrizado.

```
<typedef class="com.mycompany.usertypes.DefaultValueIntegerType" name="default_zero">
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

É possível anular os parâmetros provido em um typedef em um caso-por-caso base usando parâmetros de tipo no mapeamento de propriedade.

Embora a rica variedade de tipos prefabricados e o suporte a componentes do Hiberante siguinifique que você raramente *precise* usar um tipo personalizado; sem duvida é considerada uma boa pratica usar tipos personalizado paa classes (não entidades) que frequentemente aparecem na sua aplicação, Por exempl, uma clase `MonetaryAmount` é uma boa candidata para um `CompositeUserType`, embora pudesse ser mapeada facilmente como um componente. Uma motivação para isto é abstração. Com um tipo personalizado, seus documentos de mapeamento seriam protegidos contra possíveis mudanças no seu modo de representar valores monetários.

5.3. Mapeando uma classe mais de uma vez

É possível prover mais de um mapeamento para uma classe persistente particular. Neste caso você tem que es-

pecificar um *nome de entidade* para garantir unicidade entre as istancias das duas entidades mapeadas. (Por default, o nome de entidade está igual ao nome de classe.) O Hibernate permite especificar o nome de entidade ao trabalhar com objetos persistentes, ao escrever consular, ou ao traçar associações com a entidade mencionada.

```
<class name="Contract" table="Contracts"
        entity-name="CurrentContract">
    ...
    <set name="history" inverse="true"
            order-by="effectiveEndDate desc">
        <key column="currentContractId"/>
        <one-to-many entity-name="HistoricalContract"/>
    </set>
</class>

<class name="Contract" table="ContractHistory"
        entity-name="HistoricalContract">
    ...
    <many-to-one name="currentContract"
            column="currentContractId"
            entity-name="CurrentContract"/>
</class>
```

Observe como como são especificadas associações usando `entity-name` em vez de `class`.

5.4. SQL quoted identifiers

Você pode forçar o Hibernate a colocar entere aspas um identificador no SQL gerado incluindo o nome da tabela ou nome de coluna em backticks no documento de mapeamento. O Hibernate usará o estilo de quotation correto para o `Dialect` de SQL (normalmente aspas duplas, mas parênteses para SQL server e backticks para MySQL).

Nota do traduros: não consegui de maneira nemhuma traduzir quoted, quotation e backsticks sugestões são bem vindas

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.5. Metadata alternativos

XML não é para todo o mundo, assim há alguns modos alternativos para definir o metadata de mapeamento O/R dentro Hibernate.

5.5.1. Usando marcação XDoclet

Muitos usuários do Hibernate preferem embutir a informação de mapeamento diretamente no código usando as tags `@hibernate.tags` XDoclet . Nós não cobriremos esta abordagem neste documento, pois isto é considerada estritamente parte de XDoclet. Porém, nós incluímos o seguinte exemplo da classe de `Cat` com mapeamento XDoclet.

```
package eg;
import java.util.Set;
import java.util.Date;

```

```
/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mother;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /*
     * @hibernate.id
     *  generator-class="native"
     *  column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     *  column="PARENT_ID"
     */
    public Cat getMother() {
        return mother;
    }
    void setMother(Cat mother) {
        this.mother = mother;
    }

    /**
     * @hibernate.property
     *  column="BIRTH_DATE"
     */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     *  column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     *  column="COLOR"
     *  not-null="true"
     */
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     *  inverse="true"
```

```
     *  order-by="BIRTH_DATE"
     * @hibernate.collection-key
     *  column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }

    /**
     * @hibernate.property
     *  column="SEX"
     *  not-null="true"
     *  update="false"
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
        this.sex=sex;
    }
}
```

Veja o site do Hibernate para mais exemplos de XDoclet e Hibernate.

5.5.2. Usando anotações JDK 5.0

O JDK 5.0 introduziu anotações ao estilo XDoclet no nível de linguagem, com verificação segura de tipos e verificação em tempo de compilação. Este mecanismo é mais poderoso que anotações XDoclet e com mais suporte de ferramentas e IDEs. Por exemplo, IntelliJ IDEA suporta auto-completion e sintaxe highlighting para anotações JDK 5.0. A nova revisão da especificação de EJB (JSR-220) usa anotações JDK 5.0 como o mecanismo de metadata primário para beans de entidade. O Hibernate3 implementa o `EntityManager` da JSR-220 (a API de persistência), suporte para mapeamento de metadata está disponível pelo pacote *Hibernate Annotations*, para download separado. Ambos os metadatas, EJB3 (JSR-220) e Hibernate3 são suportados.

Este é um exemplo de uma classe POJO anotado como um EJB entity bean:

```
@Entity(access = AccessType.FIELD)
public class Customer implements Serializable {

    @Id;
    Long id;

    String firstName;
    String lastName;
    Date birthday;

    @Transient
    Integer age;

    @Embedded
    private Address homeAddress;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="CUSTOMER_ID")
    Set<Order> orders;

    // Getter/setter and business methods
```

```
}
```

Veja que o suprote a Anotações JDK 5.0 (e JSR-220) ainda está em desenvolvimento e não está completa. Por favor veja o módulo de Anotações do Hibernate para mais detalhes.

5.6. Propriedades geradas

Propriedades geradas são propriedades que têm seus valores geradas pelo banco de dados. Tipicamente, aplicações Hibernate precisaram `atualizar` os objetos que contêm qualquer propriedade para a qual o banco de dados gera valores. Marcar essas propriedades como geradas faz com que a aplicação delege esta responsabilidade para o Hibernate. Essencialmente, sempre que o Hibernate faz um SQL de INSERT ou UPDATE para a entidade que tem propriedades definidas com geradas, imediatamente após é executado um SELECT para recuperar os valores gerados.

Propriedades marcadas como geradas tambem devem ser non-insertable e non-updateable. Somente Seção 5.1.7, “version (opcional)”, Seção 5.1.8, “timestamp (opcional)”, e Seção 5.1.9, “property” podem ser marcadas como geradas.

`never` (o default) - Não há meios de se saber se determinado valor de propriedade não é gerado dentro do banco de dados.

`insert` - estados em que um determinado valor de propriedade é gerado no momento do insert, mas não é regenerado em atualizações subseqüentes. Coisas como criar uma data entrariam nesta categoria. Veja que as propriedade Seção 5.1.7, “version (opcional)” e Seção 5.1.8, “timestamp (opcional)” que podem ser marcadas como geradas, esta opção não está lá disponível...

`always` - estado em que o valor da propriedade é gerado tanto no insert como no update

5.7. Objetos auxiliares de banco de dados

Permite o CREATE e o DROP arbitrário de objetos do banco de dados, junto com as ferramentas de evolução de schema do Hibernate, prove a habilidade para definir completamente um schema dentro dos arquivos de mapeamento doHibernate. Embora especificamente projetado por criar e deleter coisas como trigger ou stored procedures, realmente qualquer comando de SQL que pode ser executado pelo metodo `java.sql.Statement.execute()` inclusive(ALTERs, INSERTS, etc). Há essencialmente dois modos por definir obejtos auxiliares de banco de dados...

O primeiro modo é listar explicitamente os comandos CREATE e DROP fora no arquivo de mapeamento:

```
<hibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</hibernate-mapping>
```

O segundo modo é prover uma classe customizada que sabe construir os comandos CREATE e DROP. Esta classe customizada tem que implementar a interface `org.hibernate.mapping.AuxiliaryDatabaseObject` .

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
```

```
    </database-object>
</hibernate-mapping>
```

Adicionalmente, estes objetos de banco de dados podem ser opcionamente escopados de maneira que eles só se apliquem quando certos dialetos forem usados.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9Dialect"/>
        <dialect-scope name="org.hibernate.dialect.OracleDialect"/>
    </database-object>
</hibernate-mapping>
```

Capítulo 6. Mapeamento de Coleções.

6.1. Persistent collections

O Hibernate requer que os campos que sejam coleções de valores persistentes sejam declarados como uma interface, por exemplo:

```
public class Product {
    private String serialNumber;
    private Set parts = new HashSet();

    public Set getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }
}
```

A interface atual poderia ser `java.util.Set`, `java.util.Collection`, `java.util.List`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap` ou... qualquer uma de sua preferencia! (Onde "qualquer uma de sua preferencia" significa que você terá que escrever uma implementação de `org.hibernate.usertype.UserCollectionType`.)

Veja como nós inicializamos a variável de uma instancia de um `HashSet`. Esse é o melhor modo para inicializar uma propriedade que seja uma coleção de valores recentemente instanciados (não-persistentes). Quando você persistir a instancia - chamando `persist()`, por exemplo - O Hibernate na verdade substituirá o `HashSet` por uma instancia de uma implementação do próprio Hibernete de`Set`. Você obtetá um erro ao fazer algo parecido com isso:

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);
kittens = cat.getKittens(); // Okay, kittens collection is a Set
(HashSet) cat.getKittens(); // Error!
```

As coleções persistentes injetadas pelo Hibernate se comportam como `HashMap`, `HashSet`, `TreeMap`, `TreeSet` ou `ArrayList` dependendo do tipo de interface.

Instancias de coleções têm o comportamento habitual de tipos de valor. Eles são persistidos automaticamente quando referenciada por um objeto persistente e deletadas automaticamente quando a referencia e excluida. Se uma coleção é passada de um objeto persistente a outro, seus elementos podem ser movidos de uma tabela para outra. Duas entidades não podem compartilhar uma referência a mesma instancia de uma coleção. Devido ao modelo de relational subjacente, propriedades que sejam coleções não suportam valores nulos; O Hibernate não distingue entre uma referência nula de coleção nula e uma coleção vazia.

Você não deve ter que preocupar muito sobre isso. Use coleções persistentes do mesmo modo você usa coleções ordinárias do Java. Apenas tenha certeza você entende a semântica de associações de bidirectional (discutido depois).

6.2. Mapeamento de coleções

O elemento Hibernate usado para mapear a coleção depende do tipo da interface. Por exemplo, um elemento `<set>` é usado para mapear propriedades do tipo `Set`.

```
<class name="Product">
    <id name="serialNumber" column="productSerialNumber"/>
    <set name="parts">
        <key column="productSerialNumber" not-null="true"/>
        <one-to-many class="Part"/>
    </set>
</class>
```

Alem do `<set>`, existem também `<list>`, `<map>`, `<bag>`, `<array>` e `<primitive-array>` elementos de mapeamento. O elemento `<map>` e presentativo:

```
<map
    name="propertyName"                                          (1)
    table="table_name"                                           (2)
    schema="schema_name"                                         (3)
    lazy="true|extra|false"                                      (4)
    inverse="true|false"                                         (5)
    cascade="all|none|save-update|delete|all-delete-orphan|delet(6)e-orphan"
    sort="unsorted|natural|comparatorClass"                      (7)
    order-by="column_name asc|desc"                              (8)
    where="arbitrary sql where condition"                        (9)
    fetch="join|select|subselect"                                (10)
    batch-size="N"                                               (11)
    access="field|property|ClassName"                            (12)
    optimistic-lock="true|false"                                 (13)
    mutable="true|false"                                         (14)
    node="element-name|."
    embed-xml="true|false"
>

    <key .... />
    <map-key .... />
    <element .... />
</map>
```

(1) `name` o nome da propriedade que é uma coleção

(2) `table` (opcional - valor default para o nome da propriedade) o nome da tabela de associação (não usado para associações one-to-many)

(3) `schema` (opcional)o nome de um schema de tabelas para sobrescrever o schema declarado no elemento raiz.

(4) `lazy` (opicional - valor default `true`) pode ser usado para desabilitar a carga posterior e especificar que a associação sempre deve ser carregada, ou habilitar "extra-lazy" onde a maioria das operações não inicializa a coleção (satisfatório para coleções muito grandes).

(5) `inverse` (opicional - valor default `false`) marca esta coleção como o fim "inverso" de uma associação bi-directional.

(6) `cascade` (opicional - valor default `none`) permite operações em cascata nas entidades filhas.

(7) `sort` (opicional) especifica que a coleção deve ser ordenada pela ordem `natural`, ou uma determinada classe de comparador.

(8) `order-by` (opicional, apenas para JDK1.4 only) especifiqua uma coluna (ou colunas) da tabela que define a ordem de intereação do `Map`, `Set` ou `Set`, junto com os opcionais `asc` ou `desc`

(9) `where` (opicional) specifica uma condição SQL `WHERE` arbitrária para ser usado na recuperação ou remoção da coleção (útil se a coleção deve conter só um subconjunto dos dados disponíveis)

(10) `fetch` (opcional, valor default `select`) Escolhe entre outer-join fetching, fetching by sequential select, ou fetching by sequential subselect.

(11) `batch-size` (opicional, valor default `1`) especifica o "tamnho do lote" para a carga das instancias desta coleção.

(12) `access` (opcional, valor default `property`): A estratégia do Hibernate usada para acessar o valor de propri-

edade de coleção.

(13) `optimistic-lock` (opcional, valor default `true`): Especifica que mudanças para o estado da coleção resultam em incremento da versão da entidade principal. (Para associações um-para-muitos, é razoável desabilitar essa opção.)

(14) `mutable` (opcional, valor default `true`): Um valor `false` especifica que os elementos da coleção nunca mudam (uma otimização de desempenho secundária em alguns casos).

6.2.1. Collection foreign keys

Instancias de coleção são distinguidas no banco de dados pela foreign key da entidade que possui a coleção. Esta foreign key referencia a *coluna chave da coleção* (ou colunas) na tabela de coleção. A coluna coleção e mapeada pelo elemento `<key>`.

Pode haver uma constraint not null na coluna foreing key. Para a maioria das coleções, isto é verdadeiro. Para uma associação unidirectional um para muitos, a coluna foreign key possui valor nulo por default, assim você pode precisar especificar `not-null="true"`.

```
<key column="productSerialNumber" not-null="true"/>
```

A constraint foreign key mais usada `ON DELETE CASCADE`.

```
<key column="productSerialNumber" on-delete="cascade"/>
```

Veja o capitulo anterior para uma definição completa do elemento `<key>`.

6.2.2. Elementos da coleção

Coleções podem conter quase qualquer tipo do Hibernate, incluindo todos os tipos básicos, tipos customizados, componentes, e é claro, referências para outras entidades. Esta é uma distinção importante: um objeto em uma coleção pode ser controlado com semântica de "valor" (seu ciclo de vida depende completamente do dono de coleção) ou poderia ser uma referência a outra entidade, com seu próprio ciclo de vida. Nesse ultimo caso, somente a "ligação" entre os dois objetos é considerada para persistencia pela coleção.

O tipo contido está chamado de *o tipo de elemento de coleção*. Elementos de coleção são mapeados por `<element>` ou `<composite-element>`, ou no caso de referências a entidades, com `<one-to-many>` ou `<many-to-many>`. Os primeiros dois elementos de mapeiam com semântica de valor, os próximos são usados dois para mapear associações de entidade.

6.2.3. Coleções indexadas

Todos os mapeamento de coleções, exceto aqueles com a semântica de set e bag, precisam de uma *coluna indexada* na tabela de coleção - a coluna para se fazer um mapeamento para um indice de array, ou um indice de uma `List`, ou uma chave de um `Map`. O índice de um `Map` pode ser de qualquer tipo básico, mapeado com `<map-key>`, também pode ser uma referência de entidade mapeada com `<map-key-many-to-many>`, ou pode ser um tipo composto, mapeado com `<composite-map-key>`. O índice de um aray ou list sempre é tipo `integer` e é mapeado usando o elemento `<list-index>`. A coluna mapeada contém uma sequencia de inteiros (inicando de zero, por default).

```
<list-index
        column="column_name"                (1)
        base="0|1|..."/>
```

(1) `column_name` (requerido): O nome da coluna que contém os valores de índice de coleção.

(1) `base` (opcional, valor default `0`): O valor da coluna de índice que corresponde ao primeiro elemento da lista ou array.

```
<map-key
        column="column_name"                (1)
        formula="any SQL expression"        (2)
        type="type_name"                    (3)
        node="@attribute-name"
        length="N"/>
```

(1) `column` (opcional): O nome da coluna que contém os valores de índice de coleção.

(2) `formula` (opcional): Uma fórmula SQL usada para avaliar a chave do mapa.

(3) `type` (requerido): O tipo das chaves de mapa.

```
<map-key-many-to-many
        column="column_name"                (1)
        formula="any SQL expression"        (2)(3)
        class="ClassName"
/>
```

(1) `column` (opcional): O nome da foreign key para os valores do índice de coleção.

(2) `formula` (optional): A SQL formula used to evaluate the foreign key of the map key. `formula` (opcional): Uma fórmula SQL usada para avaliar a foreign key da chave do map.

(3) `class` (requerido): A classe de entidade usada como a chave do map.

Se sua tabela não possui uma coluna de índice, e você ainda deseja usar uam `List` como o tipo de propriedade, você pode mapear a propriedade como um <*bag*> do Hibernate. Um bag não retém sua ordem quando é recuperado do banco de dados, mas opcionalmente pode ser ordenado ou clasificado.

Existe uma grande quantidade de mapeamentos que podem ser usados para coleções, cobrindo muitos modelos relacionais comuns. Nós sugerimos que você experimente com a ferramenta de geração de schema para ter uma ideia de como as várias declarações de mapeamento serão traduzem nas tabelas de banco de dados.

6.2.4. Coleções de valores associações muitos-para-muitos

Qualquer coleção de valores ou associação muito-para-muitos requer uma *tabela de associação* dedicada com uma ou várias colunas foreign keys, *coluna de elemento de coleção* ou colunas e possivelmente uma ou várias colunas de índice.

Para uma coleção de valores, nós usamos a tag `<element>`.

```
<element
        column="column_name"                    (1)
        formula="any SQL expression"            (2)
        type="typename"                         (3)
        length="L"
        precision="P"
        scale="S"
        not-null="true|false"
        unique="true|false"
        node="element-name"
/>
```

(1) `column` (opcional): O nome da coluna que contém os valores de elemento de coleção.

(2) `formula` Uma fórmula de SQL usada para avaliar o elemento.

(3) `type` (requerido): O tipo do elemento de coleção.

Uma associação *muitos-para-muitos* é especificada usando o elemento `<many-to-many>`.

```
<many-to-many
        column="column_name"                                (1)
        formula="any SQL expression"                        (2)
        class="ClassName"                                   (3)
        fetch="select|join"                                 (4)
        unique="true|false"                                 (5)
        not-found="ignore|exception"                        (6)
        entity-name="EntityName"                            (7)
        property-ref="propertyNameFromAssociatedClass"      (8)
        node="element-name"
        embed-xml="true|false"
    />
```

(1) `column` (opcional): O nome da coluna foreign key .

(2) `formula` (opcional): Uma fórmula de SQL usada para avaliar o valor do elemento foreign key.

(3) `class` (requerido): O nome da classe associada.

(4) `fetch` (opcional - valor default `join`): habilita a recuperação por união externa ou fetching seguencial para esta associação. Este é um caso especial; para a recuperação antecipada (em um único `SELECT`) de uma entidade e seus relacionamentos muitos-para-muitos para outras entidades, você habilitaria o `join` não só indo buscar da própria coleção, mas também os atributo `<many-to-many>` nos elementos aninhados.

(5) `unique` (opcional): Habilita a geração de DDL de unique constraint para a coluna foreign key. Isto faz com que uma associação multipla efetivamente funcione como um-para-muitos.

(6) `not-found` (opcional - default para exceção): Especifica o comportamento das foreign keys que referênciam linhas perdidas: `ignore` trata uma linha perdida como uma associação nula.

(7) `entity-name` (opcional): O nome de entidade da classe associada, como uma alternativa para `class`.

(8) `property-ref`: (opcional) O nome da propriedade da classe associada que é associada a esta foreign key. Se não for especificada, a chave primária da classe associada será usada.

Alguns exemplos, primeiro, um set de strings:

```
<set name="names" table="person_names">
    <key column="person_id"/>
    <element column="person_name" type="string"/>
</set>
```

Uma bag que contém inteiros (com uma ordem de interação determinada pelo atributo `order-by`):

```
<bag name="sizes"
        table="item_sizes"
        order-by="size asc">
    <key column="item_id"/>
    <element column="size" type="integer"/>
</bag>
```

Um array de entidades - neste caso, uma associação muitos-para-muitos:

```
<array name="addresses"
        table="PersonAddress"
        cascade="persist">
    <key column="personId"/>
    <list-index column="sortOrder"/>
    <many-to-many column="addressId" class="Address"/>
</array>
```

Um map de indices string para datas:

```
<map name="holidays"
        table="holidays"
        schema="dbo"
        order-by="hol_name asc">
    <key column="id"/>
```

```
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

Um list de componentes (discutido no próximo capítulo):

```
<list name="carComponents"
        table="CarComponents">
    <key column="carId"/>
    <list-index column="sortOrder"/>
    <composite-element class="CarComponent">
        <property name="price"/>
        <property name="type"/>
        <property name="serialNumber" column="serialNum"/>
    </composite-element>
</list>
```

6.2.5. Associações um-para-muitos

Um *associação um-para-muitos* associa as tabelas das duas classes através de uma foreign key, sem a intervenção da tabela de coleção. Este mapeamento perde parte da semântica de coleções de Java normais:

- Uma instancia da clase de entidade contida não pode pertencer a mais de uma instancia de coleção.
- Uma instancia da clase de entidade contida não pode aparecer em mais de um indice da coleção.

Uma associação de `Product` para `Part` requer existência de uma coluna foreign key e possivelmente uma coluna indixada ta tabela `Part`. A tag `<one-to-many>` indica que esta é uma associação um-para-muitos.

```
<one-to-many
        class="ClassName"                                   (1)
        not-found="ignore|exception"                        (2)
        entity-name="EntityName"                            (3)
        node="element-name"
        embed-xml="true|false"
    />
```

(1) `class` (required): The name of the associated class. `class` (requirido): O nome da classe associada.

(2) `not-found` (opicional - valor default `exception`): Especifica como deve ser manipulado os identificadores em cache que fazem referência a linhas sem associação: `ignore` tratará uma linha perdida como uma associação nula.

(3) `entity-name` (optional): The entity name of the associated class, as an alternative to `class`. entidade-nome (opcional): O nome de entidade da classe associada, como uma alternativa classificar.

Veja que o elemento `<one-to-many>` não precisa que seja declarado qualquer coluna. Também não é necessário especificar o nome da tabela em qualquer lugar.

Nota muito importante: Se a coluna foreign key de uma associação `<one-to-many>` não for declarada como`NOT NULL`, você deve declarar `<key>` como `not-null="true"` ou *usar uma associação de bidirectional* no mapeamento da coleção configurando `inverse="true"`. Veja a discussão de associações de bidirectional ainda neste capítulo.

Este exemplo mostra o mapeamento da entidades `Part` através do nome (onde `partName` é uma propriedade persistente de `Part`). Veja o uso de um índice baseado em fórmula.

```
<map name="parts"
        cascade="all">
    <key column="productId" not-null="true"/>
    <map-key formula="partName"/>
    <one-to-many class="Part"/>
```

```
</map>
```

6.3. Mapeamento avançado de coleções

6.3.1. Coleções ordenadas

O Hibernate suporta coleções que implementam `java.util.SortedMap` e `java.util.SortedSet`. Você tem que especificar um comparator no arquivo de mapeamento:

```
<set name="aliases"
            table="person_aliases"
            sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

Os valores do atributo `sort` permitidos são `unsorted`, `natural` e o nome de uma classe que implementa `java.util.Comparator`.

Coleções ordenadas na verdade se comportam como `java.util.TreeSet` ou `java.util.TreeMap`.

Se você quiser que o banco de dados ordene os elementos de coleção use o atributo `order-by` nos mapeamentos de `set`, `bag` ou `map`. Esta solução só está disponível apartir do JDK 1.4 ou superior (que é implementado usando `LinkedHashSet` ou `LinkedHashMap`). Isto executa a ordenação na query SQL, não em memória.

```
<set name="aliases" table="person_aliases" order-by="lower(name) asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

Observe que o valor do atributo `order-by` é uma ordenação SQL, e não HQL!

Associações podem ser ordenadas até mesmo por alguns critérios arbitrários em tempo de execução usando um `filter()` da coleção.

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.name" ).list();
```

6.3.2. Associações Bidirectionais

Uma *associação de bidirectional* permite navegação de ambos os "extremos" da associação. São suportados dois tipos de associação de bidirectional:

um-para-muitos

set ou bag valorados em um extremo, monovaluados no outro

muitos-para-muitos
set ou bag valorados nos dois extremos

Você pode especificar uma associação bidirectional muitos-para-muitos simplesmente mapeando dois associações muitos-para-muitos para a mesma tabela do banco de dados e declarando um dos lados como *inverse*(você pode fazer isso, mas a coleção não pode ser indexada).

Aqui é um exemplo de uma associação bidirectional many-to-many; cada categoria pode ter muitos artigos e cada artigo podem estar em muitas categorias:

```
<class name="Category">
    <id name="id" column="CATEGORY_ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="Item">
    <id name="id" column="CATEGORY_ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

As mudanças feitas somente de um lado da associação *não* são persistidas. Isto significa que o Hibernate tem duas representações na memória para cada associação bidirecional, uma associação de A para B e uma outra associação de B para A. Isto é mais fácil de compreender se você pensa sobre o modelo do objetos do Java e como nós criamos um relacionamento muitos para muitos em Java:

```

category.getItems().add(item);          // The category now "knows" about the relationship
item.getCategories().add(category);     // The item now "knows" about the relationship

session.persist(item);                   // The relationship won't be saved!
session.persist(category);               // The relationship will be saved
```

A outra ponta é usada salvar a representação em memória à base de dados.

Você pode definir uma associação bidirecional um para muitos através de uma associação um-para-muitos indicando as mesmas colunas da tabela que à associação muitos-para-um e declarando a propriedade `inverse="true"`

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <set name="children" inverse="true">
        <key column="parent_id"/>
        <one-to-many class="Child"/>
    </set>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
```

```
        column="parent_id"
        not-null="true"/>
</class>
```

Mapear apenas uma das pontas da associação com `inverse="true"` não afeta as operações em cascata, isso é um conceito ortogonal.

6.3.3. Associações bidirecionais com coleções indexadas

Uma associação bidirecional onde uma dos lados e representa pôr uma `<list>` ou `<map>` requer uma consideração especial. Se houver uma propriedade da classe filha que faça o mapeamento da coluna do índice, sem problema, pode-se continuar usando `inverse="true"` no mapeamento da coleção.

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children" inverse="true">
        <key column="parent_id"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <property name="name"
        not-null="true"/>
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

Mas, se não houver nenhuma propriedade na classe filha, não podemos ver essa associação como verdadeiramente bidirecional (há uma informação disponível em um lado da associação que não está disponível no extremo oposto). Nesse caso, nos não podemos mapear a coleção usando `inverse="true"`. Nos devemos usar o seguinte mapeamento:

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children">
        <key column="parent_id"
            not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
</class>
```

Veja que neste mapeamento, que um dos lado da associação, a coleção, é responsável pela atualização da foreign key. TODO: Isso realmente resulta em updates desnecessários ?.

6.3.4. Associações Ternárias

Há três meios possíveis de se mapear uma associação ternária. Uma é usar um `Map` com uma associação como seu índice:

```
<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

```
<map name="connections">
    <key column="incoming_node_id"/>
    <map-key-many-to-many column="outgoing_node_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

A segunda maneira é simplesmente remodelar a associação das classes da entidade. Esta é a maneira que nós usamos de uma maneira geral.

Uma alternativa final é usar os elementos compostos, que nós discutiremos mais tarde.

6.3.5. `Usando o <idbag>`

Se você concorda com nossa visão que chaves compostas são uma coisa ruim e que as entidades devem ter identificadores sintéticos (surrogate keys), então você deve estar achando um pouco estranho que as associações muitos para muitos usando coleções de valores que nós mostramos estejam mapeadas com chaves compostas! Bem, este ponto é bastante discutível; um simples tabela de associação não parece se beneficiar muito de uma surrogate key (entretanto uma coleção de valores compostos *sim*). Opcionalmente, o Hibernate prove uma maneira de mapear uma associação muitos para muitos com uma coleção de valores para uma tabela com uma surrogate key.

O elemento `<idbag>` permite mapear um `List` (ou uma `Collection` com uma semântica de bag.

```
<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>
```

Como você pode ver, um `<idbag>` possui um gerador de id sintético, igual uma classe de entidade! Uma surrogate key diferente é associada para cada elemento de coleção. Porém, o Hibernate não prove nenhum mecanismo para descobrir qual a surrogate key de uma linha em particular.

Note que o desempenho de atualização de um `<idbag>` é *much* melhor que um `<bag>` normal! O Hibernate pode localizar uma linha individual eficazmente e atualizar ou deletar individualmente, como um list, map ou set.

Na implementação atual, a estratégia de geração de identificador `native` não é suportada para identificadores de coleção usando o `<idbag>`.

6.4. Exemplos de coleções

As seções anteriores são uma grande confusão. Assim sendo vejamos uma exemplo. Essa classe:

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=children; }

    ....
    ....
}
```

tem uma coleção de instancias de `Child`. Se cada Child tiver no máximo um parent, o mapeamento natural é uma associação um para muitos:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Esse mapeamento gera a seguinte definição de tabelas

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

Se o parent for *obrigatório*, use uma associação bidirecional um para muitos:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>
```

```
    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="Parent" column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

Repare na constraint `NOT NULL`:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

Uma outra alternativa, no caso de você insistir que esta associação devea ser unidirecional, você pode declarar a constraint como `NOT NULL` na tag `<key>` do mapeamento:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id" not-null="true"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Por outro lado, se uma child puder ter os múltiplos parents, a associação apropriada é muitos-para-muitos:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Definições das tabelas:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

Para mais exemplos e um exemplo completo de mapeamento de relacionamento de mestre/detalhe, veja Capítulo 21, *Exemplo: Mestre/Detalhe*.

Até mesmo o mapeamento de associações mais exóticas são possíveis, nós catalogaremos todas as possibilidades no próximo capítulo.

Capítulo 7. Mapeamento de Associações.

7.1. Introdução

Mapeamentos de associações são freqüentemente a coisa mais difícil de se acertar. Nesta seção nós passaremos pêlos casos canônicos um pôr um, começando com mapeamentos unidirecionais e considerando os casos bidirecionais. Nos vamos usar `Person` e `Address` em todos os exemplos.

Nós classificaremos as associações pelo seu mapeamento ou a falta do mesmo, sua intervenção na tabela associativa, e pela sua multiplicidade.

O uso de foreing keys não obrigatórias não é considerada uma boa prática na modelagem de dados tradicional, assim todos nossos exemplos usam foreing keys obrigatórias. Esta não é uma exigência do Hibernate, e todos os mapeamentos funcionarão se você remover as constraints de obrigatoriedade.

7.2. Associações Unidirecionais

7.2.1. muitos para um

Uma *associação unidirecional muitos-para-um* é o tipo mais comum de associação unidirecional.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.2.2. um para um

Uma *associação unidirecional um-para-um em uma foreign key* é quase idêntica. A única diferença é a constraint unique na coluna.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>
```

```
<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

Uma *associação unidirecional um-para-um na primary key* geralmente usa um gerador de id special. (Note que nós invertemos a direção da associação nesse exemplo).

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.2.3. um para muitos

Uma *associação unidirecional um-para-muitos em uma foreing key* é um caso muito incomum, e realmente não é recomendada.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, personId bigint not null )
```

Nós achamos que é melhor usar uma tabela associativa para este tipo de associação.

7.3. Associações Unidirecionais com tabelas associativas

7.3.1. um para muitos

Uma *associação um-para-muitos unidirecional usando uma tabela associativa* e o mais comum. Note que se especificarmos `unique="true"`, estaremos modificando a cardinalidade de muitos-para-muitos para um-para-muitos.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

7.3.2. muitos para um

Uma *associação unidirecional muitos-para-um em uma tabela associativa* é bastante comum quando a associação for opcional.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
```

```
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.3.3. um para um

Uma *associação unidirecional um-para-um em uma tabela associativa* é extremamente incomum, mas possível.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

7.3.4. muitos para muitos

Finalmente, nós temos a *associação unidirecional muitos-para- muitos*.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personI
create table Address ( addressId bigint not null primary key )
```

7.4. Associações Bidirecionais

7.4.1. um para muitos / muitos para um

Uma *associação bidirecional muitos-para-um* é o tipo mais comum de associação. (Esse é o relacionamento padrão pai / filho.)

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

Se você usar uma `List` (ou outra coleção indexada), você precisa especificar a coluna `foreing key` como not null, e deixar o Hibernate administrar a associação do lado da coleção para que seja mantido o índice de cada elemento da coleção (fazendo com que o outro lado seja virtualmente inverso setando `update="false"` e `insert="false"`):

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
      <list-index column="peopleIdx"/>
      <one-to-many class="Person"/>
   </list>
</class>
```

É importante que você defina `not-null="true"` no elemento `<key>` no mapeamento na coleção se a coluna foreing keys for `NOT NULL`. Não declare como `not-null="true"` apenas um elemento aninhado `<column>`, mas sim o elemento `<key>`.

7.4.2. um para um

Uma *associação bidirecional um-para-um em uma foreign key* é bastante comum.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

Uma *associação bidirecional um para um em uma chave primária* usa um gerador de id especial.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.5. Associações Bidirecionais com tabelas associativas

7.5.1. um para muitos / muitos para um

Uma *associação bidirecional um para muitos em uma tabela associativa*. Veja que `inverse="true"` pode ser colocado em qualquer ponta associação, na coleção, ou no join.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
```

```
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

7.5.2. um para um

Uma *associação bidirecional um-para-um em uma tabela de associação* é algo bastante incomum, mas possivel.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true"
        inverse="true">
        <key column="addressId"
            unique="true"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

7.5.3. muitos para muitos

Finalmente, nós temos uma *bassociação bidirecional de muitos para muitos*.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true" table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personI
create table Address ( addressId bigint not null primary key )
```

7.6. Mapeamento de associações mais complexas

Joins de associações mais complexas são extremamente *raros*. O Hibernate torna possível tratar mapeamentos mais complexos usando fragmentos de SQL embutidos no documento de mapeamento. Por exemplo, se uma tabela com informações de dados históricos de uma conta define a coluna `accountNumber`, `effectiveEndDate` e `effectiveStartDate`, mapeadas assim:

```
<properties name="currentAccountKey">
    <property name="accountNumber" type="string" not-null="true"/>
    <property name="currentAccount" type="boolean">
        <formula>case when effectiveEndDate is null then 1 else 0 end</formula>
    </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

Então nós podemos mapear uma associação para a instância *corrente* (aquela com a `effectiveEndDate` igual a null) usando:

```
<many-to-one name="currentAccountInfo"
        property-ref="currentAccountKey"
        class="AccountInfo">
    <column name="accountNumber"/>
```

```
    <formula>'1'</formula>
</many-to-one>
```

Em um exemplo mais complexo, imagine que a associação entre `Employee` e `Organization` é mantida em uma tabela `Employment` cheia de dados históricos do trabalho. Então a associação do funcionário *recentemente* empregado (aquele com a `startDate` mais recente) deve ser mapeado desta maneira:

```
<join>
    <key column="employeeId"/>
    <subselect>
        select employeeId, orgId
        from Employments
        group by orgId
        having startDate = max(startDate)
    </subselect>
    <many-to-one name="mostRecentEmployer"
            class="Organization"
            column="orgId"/>
</join>
```

Você pode ser criativo com esta funcionalidade, mas geralmente é mais prático tratar estes tipos de casos, usando uma pesquisa HQL ou uma pesquisa por criteria.

Capítulo 8. Mapeamento de Componentes.

A noção de *componente* é reusada em vários contextos diferentes, para propósitos diferentes, pelo Hibernate.

8.1. Objetos dependentes

Um componente é um objeto contido que é persistido como um tipo de valor, não uma referência de entidade. O termo "componente" refere-se à noção de composição da orientação a objetos (não a componentes no nível de arquitetura). Por exemplo, você pode modelar uma pessoa desta maneira:

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

Agora `Name` pode ser persistido como um componente de `Person`. Note que `Name` define métodos getter e setter para suas propriedades persistentes, mas não necessita declarar nenhuma interface ou propriedades identifica-

doras.

Nosso mapeamento do Hibernate seria semelhante a isso

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- class attribute optional -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

A tabela pessoa teria as seguintes colunas `pid`, `birthday`, `initial`, `first` e `last`.

Assim como todos os tipos por valor, componentes não suportam referencias cruzadas. Em outras palavras, duas pessoas poderiam possuir o mesmo nome, mas os dois objetos pessoa poderiam conter dois objetos nome independentes, apenas com "o mesmo" por valor. A semântica dos valores null de um componente são *ad hoc*. No recarregameno do conteúdo do objeto, O Hibernate assumira que se todas as colunas do componente são null, então todo o componente é null. Isto seria o certo para a maioria dos propósitos.

As propriedades de um componente podem ser de qualquer tipo do Hibernate (collections, associações muitos-para-um, outros componentes, etc). Componentes agrupados *não* devem ser consideros um uso exótico. O Hibernate tem a intenção de suportar um modelo de objetos muito bem granulado.

O elemento `<component>` permite um subelemento `<parent>` que mapeia uma propriedade da classe componente como uma referência de volta para a entidade que a contém.

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name" unique="true">
        <parent name="namedPerson"/> <!-- reference back to the Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

8.2. Coleções de objetos dependentes

Coleções de componentes são suportadadas (ex. uma array de tipo `Name`). Declare sua coleção de componentes substituindo a tag`<element>` pela tag `<composite-element>`.

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"> <!-- class attribute required -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set>
```

Nota: se você definir um `Set` de elementos compostos, é muito importante implementar `equals()` e `hashCode()` corretamente.

Elementos compostos podem conter componentes mas não coleções. Se o seu elemento composto contiver componentes, use a tag `<nested-composite-element>` . Este é um caso bastante exótico – uma coleções de componentes que por si própria possui componentes. Neste momento você deve estar se perguntando se uma associação um-para-muitos não seria mais apropriada. Tente remodelar o elemento composto como uma entidade – mas note que mesmo pensando que o modelo Java é o mesmo, o modelo relacional e a semântica de persistência ainda serão diferentes.

Por favor, veja que um mapeamento de elemento composto não suporta propriedades capazes de serem null se você estiver usando um `<set>`. O Hibernate tem que usar cada valor das colunas para identificar um registro quando estiver deletando os objetos (não existe coluna primary key separada na tabela de elemento composto), que não é possível com valores null. Você tem que usar um dos dois, ou apenas propriedades não null em um elemento composto ou escolher uma `<list>`, `<map>`, `<bag>` ou `<idbag>`.

Um caso especial de elemento composto é um elemento composto com um elemento `<many-to-one>` aninhado. Um mapeamento como este permite você a mapear colunas extras de uma tabela de associação de muitos-para-muitos para a classe de elemento composto. A seguinte associação de muitos-para-muitos de `Order` para um `Item` onde `purchaseDate`, `price` e `quantity` são propriedades da associação:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- class attribute is optional -->
        </composite-element>
    </set>
</class>
```

Claro, que não pode ter uma referência para purchase no outro lado, para a navegação da associação bidirecional. Lembre-se que componentes são tipos por valor e não permitem referências compartilhadas. Uma classe `Purchase` simples pode estar no set de uma classe `Order`, mas ela não pode ser referenciada por `Item` no mesmo momento.

Até mesmo associações ternárias (ou quaternária, etc) são possíveis:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails class="eg.Purchase"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class>
```

Elementos compostos podem aparecer em pesquisas usando a mesma sintaxe assim como associações para outras entidades.

8.3. Componentes como índices de Map

O elemento `<composite-map-key>` permite você mapear uma classe componente como uma chave de um `Map`. Tenha certeza que você sobrescreveu `hashCode()` e `equals()` corretamente na classe componente.

8.4. . Componentes como identificadores compostos

Você pode usar um componente como um identificador de uma classe entidade. Sua classe componente deve satisfazer certos requisitos:

- Ele deve implementar `java.io.Serializable`.
- Ele deve re-implementar `equals()` e `hashCode()`, consistentemente com a noção de igualdade de chave composta do banco de dados.

Nota: no Hibernate 3, o segundo requisito não é absolutamente necessário. Mas atenda ele de qualquer forma.

Você não pode usar um `IdentifierGenerator` para gerar chaves compostas. Ao invés disso o aplicativo deve gerenciar seus próprios identificadores.

Use a tag `<composite-id>` (com elementos `<key-property>` aninhados) no lugar da declaração `<id>` de costume. Por exemplo, a classe `OrderLine` possui uma primary key que depende da primary key (composta) de `Order`.

```
<class name="OrderLine">

    <composite-id name="id" class="OrderLineId">
        <key-property name="lineId"/>
        <key-property name="orderId"/>
        <key-property name="customerId"/>
    </composite-id>

    <property name="name"/>

    <many-to-one name="order" class="Order"
            insert="false" update="false">
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-one>
    ....

</class>
```

Agora, qualquer foreign key que se referencie a tabela `OrderLine` também será composta. Você deve declarar isto em seus mapeamentos para outras classes. Uma associação para `OrderLine` seria mapeada dessa forma:

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
</many-to-one>
```

(Note que a tag `<column>` é uma alternativa para o atributo `column` por toda a parte.)

Uma associação `many-to-many` para `many-to-many` também usa a foreign key composta:

```
<set name="undeliveredOrderLines">
    <key column name="warehouseId"/>
    <many-to-many class="OrderLine">
        <column name="lineId"/>
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-many>
</set>
```

A collection de `OrderLines` em `Order` usaria:

```
<set name="orderLines" inverse="true">
    <key>
        <column name="orderId"/>
        <column name="customerId"/>
    </key>
    <one-to-many class="OrderLine"/>
</set>
```

(O elemento `<one-to-many>`, como de costume, não declara colunas.)

Se `OrderLine` possui uma collection, ela também tem uma foreign key composta.

```
<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key>   <!-- a collection inherits the composite key type -->
            <column name="lineId"/>
            <column name="orderId"/>
            <column name="customerId"/>
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class>
```

8.5. Componentes Dinâmicos

Você pode até mesmo mapear uma propriedade do tipo `Map`:

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component>
```

A semântica de um mapeamento `<dynamic-component>` é idêntica à `<component>`. A vantagem deste tipo de mapeamento é a habilidade de determinar as propriedades atuais do bean no momento de deploy, apenas editando o documento de mapeamento. A Manipulação em tempo de execução do documento de mapeamento também é possível, usando o parser DOM. Até melhor, você pode acessar (e mudar) o metamodelo configuration-time do Hibernate através do objeto `Configuration`.

Capítulo 9. Mapeamento de Herança

9.1. As três estratégias

O Hibernate suporta as três estratégias básicas de mapeamento de herança:

- tabela por hierarquia de classes
- tabela por subclasse
- tabela por classe concreta

Adicionalmente, o Hibernate suporta uma quarta, um tipo levemente diferente de polimorfismo:

- polimorfismo implícito

É possível usar diferentes estratégias de mapeamento para diferentes ramificações da mesma hierarquia de herança, e então fazer uso do polimorfismo implícito para alcançar polimorfismo através da hierarquia completa. De qualquer forma, O Hibernate não suporta a mistura de mapeamentos `<subclass>`, e `<joined-subclass>` e `<union-subclass>` dentro do mesmo elemento raiz `<class>`. É possível usar junto às estratégias tabela por hierarquia e a tabela por subclasse, abaixo do mesmo elemento `<class>`, combinando os elementos `<subclass>` e `<join>` (veja abaixo).

É possível definir mapeamentos `subclass`, `union-subclass`, e `joined-subclass` em documentos de mapeamento separados, diretamente abaixo de `hibernate-mapping`. Isso permite a você estender uma hierarquia de classes apenas adicionando um novo arquivo de mapeamento. Você deve especificar um atributo `extends` no mapeamento da subclasse, nomeando uma superclasse previamente mapeada. Nota: Anteriormente esta característica fazia o ordenamento dos documentos de mapeamento importantes. Desde o Hibernate3, o ordenamento dos arquivos de mapeamento não importa quando usamos a palavra chave extends. O ordenamento dentro de um arquivo de mapeamento simples ainda necessita ser definido como superclasse antes de subclasse.

```
<hibernate-mapping>
    <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
         <property name="name" type="string"/>
    </subclass>
</hibernate-mapping>
```

9.1.1. Tabela por hierarquia de classes

Suponha que tenhamos uma interface `Payment`, com sua implementação `CreditCardPayment`, `CashPayment`, `ChequePayment`. O mapeamento da tabela por hierarquia seria parecido com:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
```

```
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

Exactly one table is required. There is one big limitation of this mapping strategy: columns declared by the subclasses, such as `CCTYPE`, may not have `NOT NULL` constraints.

9.1.2. Tabela por subclasse

Um mapeamento de tabela por subclasse seria parecido com:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```

Quatro tabelas são necessárias. As três tabelas subclasses possuem associação de primary key para a tabela de superclasse (então o modelo relacional é atualmente uma associação de um-para-um).

9.1.3. Tabela por subclasse, usando um discriminador

Note que a implementação de tabela por subclasse do Hibernate não necessita de coluna de discriminador. Outro mapeador objeto/relacional usa uma implementação diferente de tabela por subclasse, que necessita uma coluna com o tipo discriminador na tabela da superclasse. A abordagem escolhida pelo Hibernate é muito mais difícil de implementar, porém de forma argumentável mais correto de um ponto de vista relacional. Se você deseja utilizar uma coluna discriminadora com a estratégia tabela por subclasse, você pode combinar o uso de `<subclass>` e `<join>`, dessa maneira:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
```

```
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class>
```

A declaração opcional `fetch="select"` diz ao Hibernate para não buscar os dados da subclasse `ChequePayment`, usando um outer join quando estiver pesquisando pela superclasse.

9.1.4. Misturando tabela por hierarquia de classes com tabela por subclasse

Você pode até mesmo misturar a estratégia de tabela por hierarquia e tabela por subclasse usando esta abordagem:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

Para qualquer uma dessas estratégias de mapeamento, uma associação polimórfica para a classe raiz `Payment` deve ser mapeada usando `<many-to-one>`.

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

9.1.5. Tabela por classe concreta

Existem duas formas que poderíamos usar a respeito da estratégia de mapeamento de tabela por classe concreta. A primeira é usar `<union-subclass>`..

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
```

```
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class>
```

Três tabelas estão envolvidas para as subclasses. Cada tabela define colunas para todas as propriedades da classe, incluindo propriedades herdadas.

A limitação dessa abordagem é que se uma propriedade é mapeada na superclasse, o nome da coluna deve ser o mesmo em todas as tabelas das subclasses. (Nós devemos melhorar isto em um futuro release do Hibernate). A estratégia do gerador de identidade não é permitida em união de subclasses(union-subclass) herdadas, na verdade a fonte de chave primária deve ser compartilhada através de todas subclasses unidas da hierarquia.

Se sua superclasse é abstrata, mapeie ela com `abstract="true"`. Claro, que se ela não for abstrata, uma tabela (padrão para `PAYMENT` no exemplo acima) adicional é necessária para segurar as instâncias da superclasse.

9.1.6. Tabela por classe concreta, usando polimorfismo implícito

Uma abordagem alternativa é fazer uso de polimorfismo implícito:

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

Veja que em nenhum lugar mencionamos a interface `Payment` explicitamente. Também preste atenção que propriedades de `Payment` são mapeadas em cada uma das subclasses. Se você quer evitar duplicação, considere usar entidades de XML (ex. (e.g. `[ <!ENTITY allproperties SYSTEM "allproperties.xml"> ]` na declaração do `DOCTYPE` e `&allproperties;` no mapeamento).

A desvantagem dessa abordagem é que o Hibernate não gera `UNION`s SQL quando executa pesquisas polimórficas.

Para essa estratégia, uma associação polimórfica para `Payment` geralmente é mapeada usando `<any>`.

```
<any name="payment" meta-type="string" id-type="long">
```

```
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

9.1.7. Misturando polimorfismo implícito com outros mapeamentos de herança

Ainda existe uma coisa para ser observada com respeito a este mapeamento. Desde que as subclasses sejam mapeadas em seu próprio elemento `<class>` (e desde que `Payment` seja apenas uma interface), cada uma das subclasses pode ser facilmente parte de uma outra hierarquia de herança! (E você ainda pode usar pesquisas polimórficas em cima da interface `Payment`.)

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

Mais uma vez, nós não mencionamos `Payment` explicitamente. Se nós executarmos uma pesquisa em cima da interface `Payment` – por exemplo, `from Payment` – o Hibernate retorna automaticamente instâncias de `CreditCardPayment` (e suas subclasses, desde que elas também implementem `Payment`), `CashPayment` e `ChequePayment` mas não as instâncias de `NonelectronicTransaction`.

9.2. Limitações

Existem certas limitações para a abordagem do "polimorfismo implícito" comparada com a estratégia de mapeamento da tabela por classe concreta. Existe uma limitação um tanto menos restritiva para mapeamentos `<union-subclass>`.

A tabela seguinte demonstra as limitações do mapeamento de tabela por classe concreta e do polimorfismo implícito no Hibernate.

Tabela 9.1. Features of inheritance mappings

Estratégia de Herança	muitos-para-um Polimórfico	um-para-um Polimórfico	um-para-muitos Polimórfico	muitos-para-muitos Polimórfico	load()/get() Polimórfico	Pesquisas Polimórficas	Joins polimórficos
table per class-hierarchy	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
table per subclass	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
table per concrete-class (union-subclass)	`<many-to-one>`	`<one-to-one>`	`<one-to-many>` (for `inverse="true"` only)	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
table per concrete class (implicit polymorphism)	`<any>`	*not supported*	*not supported*	`<many-to-any>`	`s.createCriteria(Payment.class).add( Restrictions.idEq(id) ).uniqueResult()`	`from Payment p`	*not supported*

Capítulo 10. Trabalhando com objetos

O Hibernate é uma solução completa de mapeamento objeto/relacional que não apenas poupa o desenvolvedor dos detalhes de baixo nível do sistema de gerenciamento do banco de dados, mas também oferece um *gerenciamento de estado* para objetos. Isto é, ao contrário do gerenciamento de `instruções` SQL em camadas de persistência JDBC/SQL comuns, uma visão natural da persistência orientada a objetos em aplicações Java.

Em outras palavras, desenvolvedores de aplicações Hibernate podem sempre pensar em relação ao *estado* de seus objetos, e não necessariamente em relação a execução de instruções SQL. Este parte é responsabilidade do Hibernate e é relevante aos desenvolvedores de aplicações apenas quando estão ajustando a performance do sistema.

10.1. Estado dos objetos no Hibernate

O Hibernate define e suporta os seguintes estados de um objetos:

- *Transient* - um objeto é transiente se ele foi instanciando usando apenas o operador `new`, e não foi associado com uma `Session` do Hibernate. Ele não terá uma representação persistente no banco de dados e nenhum identificador será atribuído para ele. Instâncias transientes serão destruídas pelo coletor de lixo se a aplicação não manter sua referência. Use uma `Session` do Hibernate para tornar o objeto persistente (e deixe o Hibernate gerenciar as instruções SQL que serão necessárias para executar esta transição).
- *Persistent* - uma instância persistente possui uma representação no banco de dados e um identificador. Ele pode ter sido salvo ou carregado, assim, ele está por definição no escopo de uma `Session`. O Hibernate irá detectar qualquer mudança feita a um objeto persistente e sincronizar o seu estado com o banco de dados quando completar a unidade de trabalho. Desenvolvedores não executam instruções manuais de `UPDATE`, ou instruções de `DELETE` quando o objeto deve ser passado para transiente.
- *Detached* - uma instância desaclopada é um objeto que foi persistido, mas sua `Session` foi fechada. A referência ao objeto continua válida, é claro, e a instância destacada desaclopada pode ser acoplada a uma nova `Session` no futuro, fazendo-o (e todas as modificações sofridas) persistente novamente. Essa característica possibilita um modelo de programação para unidades de trabalho que rodam durante muito tempo que requer um pensamento por tempo do usuário. Podemos chamar-las de *transações da aplicação*, i.e. uma unidade de trabalho do ponto de vista do usuário.

Agora iremos discutir os estados e suas transições (e os métodos do Hibernate que disparam uma transição) em mais detalhes.

10.2. Tornando os objetos persistentes

Instâncias recentemente instanciadas de uma classe persistente são consideradas *transientes* pelo Hibernate. Podemos tornar uma instância transiente em *persistente* associando-a a uma sessão:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

Se `Cat` possui um identificador gerado, o identificador é gerado e atribuído a `cat` quando `save()` for chamada. Se `Cat` possuir um identificador `Associado`, ou uma chave composta, o identificador deve ser atribuído à ins-

tância de cat antes que save() seja chamado. Pode-se usar também persist() ao invés de save(), com a semântica definada no novo esboço do EJB3.

Alternativamente, pode-se atribuir o identificador usando uma versão sobrecarregada de save().

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

Se o objeto persistido possuir objetos associados (e.g. a coleção kittens no exemplo anterior), esses objetos podem ser tornar persistente em qualquer ordem que se queira ao menos que se tenha uma restrição NOT NULL em uma coluna foreign key. Nunca há risco de violação de restrições da foreign key. Assim, pode-se violar uma restrição NOT NULL se save() for usada nos objetos em uma ordem errada.

Geralmente você não deve se importar com esses detalhes, muito provavelmente se usará a característica de *persistência transitiva* do Hibernate para salvar os objetos associados automaticamente. Então, enquanto uma restrição NOT NULL não ocorrer – Hibernate tomará conta de tudo. Persistência transitiva será discutida futuramente nesse capítulo.

10.3. Carregando o objetos

O método load() de uma Session nos fornece um meio para recuperar uma instância persistente se o identificador for conhecido. load() recebe uma classe do objeto e carregará o estado em uma instância mais recente dessa classe, no estado persistente.

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );
```

Alternativamente, você pode carregar um estado em uma instância dada:

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

Repare que load() irá lançar uma exceção irrecuperável se não houver na tabela no banco de dados um registro que combine. Se a classe for mapeada com um proxy, load() simplesmente retorna um proxy não inicializado e realmente não chamará o banco de dados até que um método do proxy seja invocado. Esse comportamento é muito útil se você deseja criar uma associação com um objeto sem que realmente o carregue do bando de dados. Isto também permite que sejam carregadas múltiplas instâncias como um grupo se batch-size estiver para o mapeamento da classe.

Se você não tiver certeza da existencia do registro no banco, você deve usar o metodo get(), que consulta o banco imediantamente e retorna um null se não existir o registro.

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
```

```
return cat;
```

Também pode-se carregar um objeto usando `SELECT ... FOR UPDATE`, usando um `LockMode`. Veja a documentação da API para maiores informações.

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

Veja que todas as instancias associadas ou coleções contidas *não* são selecionadas pelo `FOR UPDATE`, a menos que você decida especificar `lock` ou `all` como o estilo da cascata para a associação.

O recarregamento de um objeto e todas as suas coleções é possível a qualquer momento, usando o método `refresh()`. Util quando as triggers do banco de dados são usados para inicializar algumas propriedades do objeto.

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

Geralmente uma importante questão aparece neste ponto: O quanto Hibernate carrega do banco de dados e quantos SQL `SELECT` ele irá usar? Isto depende da estratégia de *recuperação* usada e explicada na Seção 19.1, “Estratégias de Fetching”.

10.4. Consultando

Se o identificador do objeto que se está buscando não for conhecido, uma consulta será necessária. O Hibernate suporta uma linguagem de consulta (HQL) orientada a objetos fácil mas poderosa. Para criação via programação de consultas, o Hibernate suporta características sofisticadas de consulta por Critério e Exemplo (QBCe QBE). Pode-se também expressar a consulta por meio de SQL nativa do banco de dados, com suporte opcional do Hibernate para conversão do conjunto de reultados em objetos.

10.4.1. Executando consultas

Consultas HQL e SQL nativa são representadas por uma instância de `org.hibernate.Query`. Esta interface oferece métodos para associação de parâmetros, tratamento de conjunto de resultados, e para a execução de consultas reais. Você pode obter uma `Query` usando a `Session` atual:

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

Geralmente uma consulta é executada ao invocar `list()`, o resultado da consulta será carregado completamente em uma coleção na memória. Instâncias de entidades recuperadas por uma consulta estão no estado persistente. O `uniqueResult()` oferece um atalho se você souber de previamente que a consulta retornará apenas um único objeto. Repare que consultas que fazem uso de buscas de coleções de forma ansiosa (eager) geralmente retornam duplicatas dos objetos raiz (mas com suas coleções inicializadas). Pode-se filtrar estas duplicatas através de um simples `Set`.

10.4.1.1. Interagindo com resultados

Ocasionalmente, deves-se ser capaz de atingir performances melhores com a execução de consultas usando o método `iterate()`. Geralmente isso será o caso esperado apenas se as instâncias dos entidades reais retornadas pela consulta já estiverem na sessão ou no caché de segundo nível. Caso elas ainda não tenham sido armazenadas, `iterate()` será mais devagar do que `list()` e pode ser necessário vários acessos ao banco de dados para um simples consulta, geralmente *1* para a seleção inicial que retorna apenas identificadores, e *n* consultas adicionais para inicializar as instâncias reais.

```
// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeliness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

10.4.1.2. Consultas que retornam tuplas

Algumas vezes as consultas do Hibernate retornam tuplas de objetos, nesse caso cada tupla é retornada como um array:

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.mother mother")
            .list()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten  = tuple[0];
    Cat mother  = tuple[1];
    ....
}
```

10.4.1.3. Resultados escalares

Consultas devem especificar uma propriedade da classe na clausula `select`. Elas também podem chamar funções SQL de agregação. Propriedades ou agregações são considerados resultados agregados (e não entidades no estado persistente).

```
Iterator results = sess.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color")
        .list()
        .iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
```

```
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

10.4.1.4. Bind de parametros

Os metodos da `Query` são fornecidos para aceitar binding de valores para parametros nomeados ao estilo JDBC `?`. *Ao contrário do JDBC, o Hibernate numera os parâmetros a partir zero.* Parâmetros nomeados são identificados na forma `:name` na string da query. As vantagens de parâmetros nomeados são:

- parâmetros nomeados são insensíveis à ordem que eles aparacem na string da query
- eles podem aparecer muitas vezes na mesma query
- eles estão auto-documentandos

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

10.4.1.5. Paginação

Se você precisa especificar limites em seu resultset (o número de máximo de linhas você quer recuperar / ou a primeira linha que você quer recuperar) você deve usar métodos da interface `Query`:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

O Hibernate sabe traduzir esses limites da query no SQL nativo de seu DBMS.

10.4.1.6. Paginação Interativa

Se seu driver JDBC suporta `ResultSets` pagináveis, a interface `Query` pode ser usada para obter um objeto `ScrollableResults` que permite uma navegação mais flexível dos resultados de consulta.

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                           "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list of cats by name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
```

```
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );

}
cats.close()
```

Veja que uma conexão de banco de dados aberta (e um cursor) é requerida para esta funcionalidade, use `setMaxResult()`/ `setFirstResult()` se você precisar de funcionalidade de paginação offline.

10.4.1.7. Externalanzo consultas nomeadas

Você pode definir consultas nomeadas no documento de mapeamento. (Lembre-se de usar uma seção CDATA se sua consulta contiver caráteres que podem ser interpretados como marcação.)

```
<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

Usando binding de parâmetros de maneira programática:

```
Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

Veja que o código do programa atual é independente da linguagem de consulta usada, você também pode definir consultas em SQL nativo no metadata, ou migrar consultas existentes para o Hibernarte colocando-as no arquivo de mapemamento.

Veja também que uma declaração de consulta dentro de um elemento `<hibernate-mapping>` requer um nome único para a consulta, enquanto uma declaração de consulta dentro de um elemento `<class>` é automaticamente único através do nome completamente qualificado da classe, por exemplo eg.Gato.ByNameAndMaximumWeight.

10.4.2. Filtrando coleções

Um *filter* de coleção é um tipo especial de consulta que pode ser aplicada a uma coleção persistente ou arrray. A string da consulta pode se referencia a `this`, significando o elemento atual da coleção .

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
);
```

A coleção retornada é considerada um bag, e é uma cópia da coleção fornecida. A coleção original não é modificada (isto vai contra da implicação do nome "filtro", mas éconsistente com comportamento esperado).

Observe que os filtros não requerem uma cláusula `from` (embora possam ser usados caso se deseje). Os filtros não estão limitados ao retorno de elementos de coleção.

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue")
    .list();
```

Mesmo uma query de filtro vazio é útil, por exemplo carregar um subconjunto dos elementos em uma coleção enorme:

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
    .setFirstResult(0).setMaxResults(10)
    .list();
```

10.4.3. Consultas por criterios

O HQL é extremamente poderoso mas alguns desenvolvedores preferem criar queries dinamicamente, usando um API orientada ao objeto, em vez de construir strings contendo as queries. O Hibernate provê uma API intuitiva de consulta `Criteria` para estes casos:

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

A API `Criteria` ae a API associada `Example` são discutidos COM mais detalhe no Capítulo 15, *Consultas por critérios*..

10.4.4. Consultas em sql nativo

Você pode criar uma consulta em SQL, usando `createSQLQuery()` e deixar que o Hibernate cuide do mapeamento do conjunto de resultados para objetos. Veja que você pode a qualquer momento chamar `session.connection()` e usar a `Connection` do JDBC diretamente. Se você escolher usar a API do Hibernate, você deve incluir os aliases SQL em chaves:

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

Consultas SQL podem conter parâmetros nomeados e posicionais, assim como as queries do Hibernate. Mais informação sobre consultas SQL nativa no Hibernate pode ser achado em Capítulo 16, *SQL nativo*.

10.5. Modificando objetos persistentes

As *Instancias persistentes transacionais* (ex. objetos carregados, salvos, criados ou recupedos por uma `Session`) pode ser manipuladas pela aplicação e qualquer mudança no estado persistente será persistido quando a `Session` for *limpa (flushed)* (discutido mais adiante neste capítulo). Não há necessidade de sechamar um método em particular (como `update()`, que tem um propósito diferente) para persistir as modificações. De modo que a forma mais direta de atualizar o estado de um objeto é carrega-lo com `load()`, e então manipula-lo diretamente, enquanto a `Session` estiver aberta:

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // changes to cat are automatically detected and persisted
```

Às vezes este modelo de programação é ineficiente visto que requer um SQL `SELECT` (para carregar um objeto) e um SQL `UPDATE` (para persistir seu estado atualizado) na mesma sessão. Então o Hibernate oferece uma maneira alternada, usando instancias destacadas.

Note que o Hibernate não ofereçe API própria para execução direta de sentenças `UPDATE` ou `DELETE`. O Hibernate é um serviço de administração de estado, *você não tem que pensar em `sentenças` para poder usar-lo. O JDBC é uma API perfeita para executar senteças SQL, você pode obter uma `Connection` JDBC a qualquer momento chamando `session.connection()`. Além disso, a noção de operações em grande volume conflitam com o mapeamento objeto / relational em aplicações online orientadas ao processamento de transações. Versões futuras do Hibernate podem prover funções de operação de massa especiais. Veja Capítulo 13, Processamento de lotes alguns truques de operações em lote.*

10.6. Modificando objetos destacados

Muitas aplicações precisam recupear um objeto em uma transação, enviar à camada de UI para manipulação, e então salvar as mudanças em uma nova transação. Aplicações que usam este tipo de abordagem em um ambiente de alta concorrencia normalmente usam dados versionados para assegurar o isolamento para a unidade "longa" de trabalho.

O Hibernate suporta este modelo provendo a reassociação de instancias destacados usando os metodos `Session.update()` ou `Session.merge()`:

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

Se o `Cat` com o identificador `catId` já tivesse sido carregado através de `secondSession` quando a aplicação tentasse reassocia-lo isto, uma exceção teria sido lançada.

Use `update()` se você estiver seguro que a sessão não contém nemhuma instancia persistente com o mesmo identificador, e `merge()` se você quiser mesclar suas modificações a qualquer momento sem considerar o estado da sessão. Em outrs palavras, o método `update()` normalmente é o primeiro que você chamaria em uma sessão recentemente autalizada, assegurando a reassociação de suas instancias desassociadas é a primeira operação a ser executada.

A aplicação deve atualizar individualmente cada instancia desassociada acessivel pela instancia desassociada

chamando o método `update()`, se e *somente* se desejar quer seus estados sejam também atualizados. Claro que isto pode ser automatizado, *persistencia transitiva*, veja Seção 10.11, “Persistência transitiva”.

O método `lock()` permite uma aplicação reassocie um objeto em uma nova sessão . Porém, a instancia desassocaida tem que estar inalterada!

```
//just reassociate:
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
sess.lock(pk, LockMode.UPGRADE);
```

Veja que `lock()` pode ser usado com vários `LockModes`, veja a documentação da API e o capítulo sobre controle de transação para mais informação. Reassociar não é o único case onde se usa `lock()`.

São discutidos outros modelos para unidades longas de trabalho em Seção 11.3, “Controle de concorrência otimista”.

10.7. Detecção automática de estado

Usuários do Hibernate pediram um método de propósito geral que salve qualquer instancia transiente através da geração um novoidentificador ou atualize / reassocie instancias desassociadas com seu identificador atual. O método `saveOrUpdate()` implementa esta funcionalidade.

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat);   // update existing state (cat has a non-null id)
secondSession.saveOrUpdate(mate);  // save the new instance (mate has a null id)
```

O uso e a semântica do método `saveOrUpdate()` parecem estar confundindo usuários novos. Primeiramente, você não deve tentar usar instancias de uma sessão em outra nova sessão, você não deveria precisar usar `update()`, `saveOrUpdate()`, ou `merge()`. Algumas aplicações nunca usarão qualquer um destes métodos.

Normalmente `update()` ou `saveOrUpdate()` são usados no seguinte cenario:

- a aplicação carrega um objeto na primeira sessão
- o objeto é passado até a camada de apresentação, a UI
- algumas modificações são feitas no objeto
- o objeto é devolvido para a camada de negocio
- a aplicação persiste estas modificações chamando `update()` em uma segunda sessão

`saveOrUpdate()` faz o seguinte:

- se o objeto já for persistente nesta sessão, não faça nada
- se outro objeto associado com a sessão tem o mesmo identificador, lance uma exceção
- se o objeto não tiver um valor na propriedade indentificadora, faça uma chamada a `save()`
- se o identificador do objeto tem o valor atribuido a um novo objeto instanciado recentemente, faça uma chamada a `save()`
- se o objeto for versionado (através de `<version>` ou `<timestamp>`), e o valor de propriedade de versão for o

mesmo valor atribuido a um novo objeto instanciado, , faça uma chamada a `save()`
- caso contrário faça uma chamada a `update()`

o `merge()` é muito diferente:

- se houver uma instancia persistente com o mesmo identificador associado a sessão, faça uma copia do estado deste objeto sobre o exemplo persistente
- se não houver nenhuma instancia persistente associado com a sessão atual, tente carrega-la do banco de dados, ou crie uma nova instancia persistento
- a instancia persistente é retornada
- o instancia retorna não estará associado com a sessão, permanece desassociada

10.8. Deletando objetos persistentes

`Session.delete()` removerá o estado de um objeto do banco de dados. Claro que, sua aplicação ainda poderia ter uma referência ao objeto deletado. É melhor pensar que `delete()` faz com que uma instancia persistente se torne transiente.

```
sess.delete(cat);
```

Pode-se deletar objetos em qualquer ordem que queria, sem risco de violar constraints foreign key. Mas mesmo assim ainda é possível a violação de uma constraint `NOT NULL` em uma coluna foreign key ao se deletar objetos na ordem errada, por exemplo se você apagar o mestre, mas esquece de apagar os detalhes.

10.9. Replicando objtos entre base de da dados diferentes

Ocasionalmente pode ser útil poder levar um gráfo de instancias persistentes e os fazer persistente em um banco de dados diferente, sem regerar os valores dos identificadores.

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

O `ReplicationMode` determina como `replicate()` tratará os conflitos entros os registros existentes no banco de dados.

- `ReplicationMode.IGNORE` - ignora o objeto quando houver um registro com o mesmo identificador no banco de dados
- `ReplicationMode.OVERWRITE` - sobreescreve qualquer registro existente com o mesmo identificador no banco de dados
- `ReplicationMode.EXCEPTION` - lança uma exceção se houver um registro com o mesmo identificador no banco de dados
- `ReplicationMode.LATEST_VERSION` - - sobreescreve o registro se o número de versão é mais novo que o número de versão do registro existente, ou caso contrário ignora

Os casos de uso para esta funcionalidade incluem a conciliação de dados inseridos em banco de dados diferentes, atualização de informações de configuração de sistema durante uma atualização de produto, desfazer mudanças feitas durante transações que não sejam ACID e mais.

10.10. Limpando a Session

De vez em quando a `Session` executará sentenças SQL necessárias para sincronizar o estado de conexão JDBC com o estado de objetos contidos memória. Este processo, o *flush*, acontece por default nos seguintes pontos

- antes da execuções de algumas queries
- a partir de `org.hibernate.Transaction.commit()`
- a partir de `Session.flush()`

As sentenças SQL são emitidas na seguinte ordem

1. todas as inserções de entidade, na mesma ordem os objetos correspondentes foram salvos usando `Session.save()`
2. toda as atualizações de entidade
3. todas as deleções de coleção
4. todas as deleções, atualizações e inserções de elemento de coleção
5. todas as inserções de coleção
6. todas as deleções de entidades, na mesma ordem em que os objetos correspondentes foram deletados usando-se `Session.delete()`

(Uma exceção são os objetos que usam geração de ID `native`, que são inseridos no momento que são salvos)

Exceto quando você chama explicitamente `flush()`, não há nenhuma garantia sobre *quando* a `Session` executa as chamadas JDBC, somente a *ordem* na qual eles são executados. Porém, o Hibernate garante que `Query.list(..)` nunca retornará dados obsoletos; nem dados errados.

É possível mudar o comportamento default de forma que o flush aconteca com menos freqüencia. A classe `FlushMode` define três modos diferentes: somente na hora do commit (e somente quando a API `Transaction` do Hibernate é usada), automaticamente usando a rotina explicada acima, ou nunca executar a menos que `flush()` seja chamado explicitamente. O último modo é útil para unidades de trabalho longas onde uma `Session` é mantida aberta e desconectada por muito tempo (veja Seção 11.3.2, “Sessão estendida e versionamento automático”).

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // allow queries to return stale state

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

Durante o flush, pode acontecer uma exceção (por exemplo se uma operação de DML violar uma constraint). O controle de exceções implica na compreensão do comportamento transacional do Hibernate, nós discutimos isto no Capítulo 11, *Transações e Concorrência*.

10.11. Persistência transitiva

É bastante incômodo salvar, apagr, reassociar objetos individuais, especialmente se você lida com um gráfo de objetos associados. Um caso comum é uma relação de mestre / detalhe. Considere o exemplo seguinte:

Se os detahes em uma relação mestre / detalhe fossem tipos de valor (por exemplo uma coleção de endereços ou strings), o ciclo de vida destes dependeriam do mestre e nenhuma ação adicional seria necessária para "cascatear" as mudanças de estado. Quando o mestre for salvo, os objetos detalhes serão salvos , quando o mestre for deletado, os detalhes tambés serão deletados, etc. Isto funciona até mesmo para operações como a remoção de um detalhe da coleção; O Hibernate detectará isto e, desde que objetos sejam tipos de valores não possam compartilhar referências, removerá os detalhes do banco de dados.

Agora considere o mesmo cenário com mestre e detalhes com sendo entidades, não tipos de valores (por exemplo categorias e item, ou gatos pais e filhos). Entidades têm seu próprio ciclo de vida, suportando o compartilhamento de referências (assim, remover uma entidade da coleção não significa que ela deva ser deletada), e por default não existe cascateamento de estado de uma entidade para qualquer outra entidade associada. O Hibernate não implementa persistência *persistencia por alcance*.

Para cada operação básica na sessão do Hibernate - incluindo `persist()`, `merge()`, `saveOrUpdate()`, `delete()`, `lock()`, `refresh()`, `evict()`, `replicate()` - há um estilo de cascateamento correspondente. Respectivamente, os estilos de cascata são `create`, `merge`, `save-update`, `delete`, `lock`, `refresh`, `evict`, `replicate`. Se você quiser cascatear uma operação ao longo de uma associação, você tem que indicar isso no documento de mapeamento. Por exemplo:

```
<one-to-one name="person" cascade="persist"/>
```

Estilos de cascateamento podem ser combinados

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

Você pode usar `cascade="all"` para especificar que *todas* as operações devem ser cascateadas ao longo da associação. O padrão `cascade="none"` especifica que nenhuma operação será cascateada.

Um estilo especialde cascata, `delete-orphan`, só aplica a associações, um-para-muitos e indica que a operação `delete()` deve ser aplicada a qualquer objeto detalhe que for removido da associação.

Recomendações:

- Normalmente não faz sentido habilitar cascata em associações `<many-to-one>` ou `<many-to-many>`. Cascateamento til para associações `<one-to-one>` e `<one-to-many>`.
- Se o período de vida do objeto detalhe é associado pelo período de vida do do objeto de pai faça disso o *ciclo de vida do objeto* especificando `cascade="all,delete-orphan"`.
- Caso contrário, você pode não precisar de tratamento decascata. Mas se você acha que irá trabalhar freqüentemente com mestres e detalhes juntos na mesma transação, e você quer se poupar de alguma código, considere usar `cascade="persist,merge,save-update"`.

Mapear uma associação (uma associação um-para-um, ou uma coleção) com `cascade="all"` marca a associação como uma relação *mestre/detalhe* onde save/update/delete no mestre resulta em save/update/delete no detalhe ou nos detalhes.

Dese jeito, uma mera referência a um detalhe por parte um mestre persistente resultará em salvar/atualizar o detalhe. Esta metáfora está incompleta, porém. Um detalhe que deixa de ser referenciado pelo mestre *não* é automaticamente deletado, exeto no caso de uma associação `<one-to-many>` mapeada com `casca-`

de="delete-orphan". A semântica precisa de cascateamento operações para uma relação de mestre / detalhe é a seguinte:

- Se um metre é passado para `persist()`, todos os detalhes são passadas para `persist()`
- Se um mestre é passado para `merge()`, todas os detalhes são passadas para `merge()`
- Se um mestre é passado para `save()`, `update()` ou `saveOrUpdate()`, todos os detalhes são passadas para `saveOrUpdate()`
- Se um detalhe transiete ou dessacoicada se tornar referenciada por um mestre persistente, ele é passado para `saveOrUpdate()`
- Se um mestre é apagado, todas as crianças são passadas para `delete()`)
- Se um detalhe é desreferenciado por um mestre persistente, nada de especial acontece - a aplicação deveria apagar o detalhe explicitamente se necessário - a menos no caso de `cascade="delete-orphan"` em que todo detalhe "órfãi" é apagada.

Finalmente, veja que o cascateando de operações pode ser aplicado a um gráfo de objeto em *call time* ou em *flush time*. Todas as operações, se habilitado, é cascateado a entidades associadas alcançávis quando a operação é executada. Porém, `save-upate` e `delete-orphan` são transitivos para todas as entidades associadas alcançável durante o flush da `Session`..

10.12. Usando metadados

O Hibernate requer um modelo muito rico a nível de metadados de todas as entidades e tipos de valores. De tempos em tempos, este modelo é muito útil à própria aplicação. Por exemplo, a aplicação pode usar os metadados do Hibernate que executa um algoritmo "inteligente" que compreende quais objetos podem ser copiados (por exemplo, tipos de valores mutáveis) ou não (por exemplo, tipos de valores imutáveis e, possivelmente, entidades associadas).

O Hibernate expõe o metadados via interfaces `ClassMetadata` e `CollectionMetadata` e pela hierarquia `Type`. Instâncias das interfaces de metadados podem ser obtidas a partir do `SessionFactory`.

```
Cat fritz = ......;
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// get a Map of all properties which are not collections or associations
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

Capítulo 11. Transações e Concorrência

O ponto o mais importante sobre o Hibernate e o controle de concorrência é que é muito fácil de ser compreendido. O Hibernate usa diretamente conexões de JDBC e recursos de JTA sem adicionar nenhum comportamento de bloqueio a mais. Nós altamente recomendamos que você gaste algum tempo com o JDBC, o ANSI e a especificação de isolamento de transação de seu sistema de gerência da base de dados.

O Hibernate não bloqueia objetos na memória. Sua aplicação pode esperar o comportamento tal qual definido pelo nível de isolamento de suas transações de banco de dados. Note que graças ao `Session`, que também é um cache de escopo de transação, o Hibernate fornece leituras repetíveis para procurar por identificadores e consultas de entidade (não pesquisas de relatórios que retornam valores escalares).

Além do versionamento para o controle automático de concorrência otimista, o Hibernate oferece também uma API (menor) para bloqueio pessimista de linhas usando a sintaxe `SELECT FOR UPDATE`. O controle de concorrência otimista e esta API são discutidos mais tarde neste capítulo.

Nós começamos a discussão do controle de concorrência no Hibernate com a granularidade do `Configuration`, `SessionFactory`, e `Session`, além de transações de base de dados e conversações longas.

11.1. Session e escopos de transações

Um `SessionFactory` é um objeto threadsafe compartilhado por todas as threads da aplicação que consome muitos recursos na sua criação. É criado uma unica vez no inicio da execução da aplicação a partir da instância de uma `Configuration`.

Uma `Session` é um objeto de baixo custo de criação, não é threadsafe, deve ser usado uma vez, para uma única requisição, uma conversação, uma única unidade do trabalho e então deve ser descartado. Um `Session` não obterá um JDBC `Connection` (ou um `Datasource`) a menos que necessite, conseqüentemente não consome nenhum recurso até ser usado.

Para completar, você também tem que pensar sobre as transações de base de dados. Uma transação tem que ser tão curta quanto possível, para reduzir a disputa pelo bloqueio na base de dados. Transações longas impedirão que sua aplicação escale a carga altamente concorrente. Por isso, em um projeto raramente é para manter uma transação de base de dados aberta durante o tempo que o usuário pensa, até que a unidade do trabalho esteja completa.

Qual é o escopo de uma unidade de trabalho? Pode uma únicoa `Session` do Hibernate gerenciar diversas transações ou é esta um o relacionamento um-para-um dos escopos? Quando deve você abrir e fechar uma `Session` e como você demarca os limites da transação?

11.1.1. Unidade de trabalho

Primeiro, não use o antipattern *sessão-por-operação*, isto é, não abra e não feche uma `Session` para cada simples chamada ao banco de de dados em uma única thread! Naturalmente, o mesmo é verdadeiro para transações. As chamadas a banco de dados em uma aplicação são feitas usando uma seqüência planejada, elas são agrupadas em unidades de trabalho atômicas. (Veja que isso também significa que um auto-commit depois de cada sentença SQL é inútil em uma aplicação, esta modalidade é ideal para o trabalho ad hoc do console do SQL. O Hibernate impede, ou espera que o servidor de aplicação impessa isso, o uso da modalidade de auto-commit.) As transações nunca são opcionais, toda a comunicação com um banco de dados tem que ocorrer dentro de uma transação, não importa se você vai ler ou escrever dados. Como explicado, o comportamento auto-commit para leitura de dados deve ser evitado, como muitas transações pequenas são improváveis de executar

melhor do que uma unidade claramente definida do trabalho. A última opção também muito mais manutenível e extensível.

O pattern mais comum em uma aplicação multi-usuário cliente/servidor é *sessão-por-requisição*. Neste modelo, uma requisição do cliente é enviada ao servidor (onde a camada de persistência do Hibernate roda), uma `Session` nova do Hibernate é aberta, e todas as operações da base de dados são executadas nesta unidade do trabalho. Logo que o trabalho for completado (e a resposta para o cliente for preparada), a sessão é descarregad e fechada. Você usaria também uma única transação de base de dados para servir às requisições dos clientes, começando e commitando-o quando você abre e fecha a `Session`. O relacionamento entre os dois é um-para-um e este modelo é um ajuste perfeito para muitas aplicações.

O desafio encontra-se na implementação. O Hibernate fornece gerência integrada da "sessão atual" para simplificar este pattern. Tudo que você tem que fazer é iniciar uma transação quando uma requisição tem que ser processada e termina a transação antes que a resposta seja enviada ao cliente. Você pode fazer onde quiser, soluções comuns são `ServletFilter`, interceptador AOP com um pointcut (ponto de corte) nos métodos de serviço ou em um container de proxy/interceptação. Um container de EJB é uma maneira padronizada para implementar aspectos cross-cutting tais como a demarcação da transação em EJB session beans, declarativamente com CMT. Se você se decidir usar demarcação programática de transação, de preferencia a API `Transaction` do Hibernate mostrada mais adiante neste capítulo, para fácilidade no uso e portabilidade de código.

Seu código de aplicação pode acessar a "sessão atual" para processar a requisição fazendo uma chamada simples a `sessionFactory.getCurrentSession()` em qualquer lugar e com a frequencia necessária. Você sempre conseguirá uma `Session` limitada a transação atual. Isto tem que ser configurado para recurso local ou os ambientes JTA. Veja Seção 2.5, “Sessões contextuais”.

Às vezes é conveniente estender o escopo de uma `Session` e de uma transação do banco de dados até que a "visão esteja renderizada". É especialmente útil em aplicações servlet que utilizam uma fase de rendenderização separada depois que a requisição ter sido processada. Estendendo a transação até que renderização da visão esteja completa é fácil de fazer se você implementar seu próprio interceptador. Entretanto, não se pode fazer facilmente se você confiar em EJBs com transações gerenciadas por contêiner, porque uma transação será terminada quando um método de EJB retornar, antes da renderização de toda visão puder começar. Veja o website e o fórum do Hibernate para dicas e exemplos em torno deste pattern *Open Session in View*.

11.1.2. Longas conversações

O pattern sessão-por-requisição não é o único conceito útil que você pode usar ao projetar unidades de trabalho. Muitos processos de negócio requerem uma totalidade de séries de interações com o usuário intercaladas com acessos a uma base de dados. Em aplicações web e corporativas não é aceitável para uma transação atrapalhe uma interação do usuário. Considere o seguinte exemplo:

- A primeira tela de um diálogo abre os dados carregado pelo usuário em através de `Session` e transação particulares. O usuário está livre modificar os objetos.
- O usuário clica em "Salvar" após 5 minutos e espera suas modificações serem persistidas; espera também que ele era a única pessoa que edita esta informação e que nenhuma modificação conflitante possa ocorrer.

Nós chamamos esta unidade de trabalho, do ponto da visão do usuário, executando uma longa *conversação* (ou *transação da aplicação*). Há muitas maneiras de você pode implementar em sua aplicação.

Uma primeira implementação simples pode manter a`Session` e a transação aberta durante o tempo de interação do usuário, com bloqueios na base de dados para impedir a modificação concorrente e para garantir o isolamento e a atomicidade. Esse é naturalmente um anti-pattern, desde que a disputa do bloqueio não permitiria o escalonameneto da aplicação com o número de usuários concorrentes.

Claramente, nós temos que usar diversas transações para implementar a conversação. Neste caso, Manter o isolamento dos processos de negócio torna-se responsabilidade parcial da camada da aplicação. Uma única conversação geralmente usa diversas transações. Ela será atômica se somente uma destas transações (a última) armazenar os dados atualizados, todas as outras simplesmente leram os dados (por exemplo em um diálogo do estilo wizard que mede diversos ciclos de requisição/resposta). Isto é mais fácil de implementar do que pode parecer, especialmente se você usar as características do Hibernate:

- *Versionamento automático* - O Hibernate pode fazer o controle automático de concorrência otimista para você, ele pode automaticamente detectar se uma modificação concorrente ocorreu durante o tempo de interação do usuário. Geralmente nós verificamos somente no fim da conversação.
- *Detached Objects*- se você se decidir usar o já discutido pattern *session-per-request*, todas as instâncias carregadas estarão no estado destacado durante o tempo em que o usuário estiver pensando. O Hibernate permite que você reatache os objetos e persita as modificações, esse pattern é chamado *session-per-request-with-detached-objects*. É usado versionamento automatico para isolar as modificações concorrentes.
- *session-per-conversation* e faz o reatamento uniforme desnecessário. Versionamento automático é usado para isolar modificações concorrentes e a *session-per-conversation* usualmente não é permitido para ser nivelado automaticamente, e sim explicitamente.

Ambos *session-per-request-with-detached-objects* e *session-per-conversation* possuem vantagens e desvantagens, nos discutiremos mais tarde neste capítulo no contexto do controle de concorrência otimista.

11.1.3. Considerando a identidade do objeto

Uma aplicação pode acessar concorrentemente o mesmo estado persistente em duas `Session`s diferentes. Entretanto, uma instância de uma classe persistente nunca é compartilhada entre duas instâncias `Session`. Por tanto, há duas noções diferentes da identidade:

Identidade da base de dados

```
foo.getId().equals( bar.getId() )
```

Identidade da JVM

```
foo==bar
```

Então para os objetos acoplados a um `Session` em `particular` (isto é no escopo de um `Session`), as duas noções são equivalentes e a identidade da JVM para a identidade da base de dados é garantida pelo Hibernate. Entretanto, quando a aplicação pode acessar concorrentemente o "mesmo" objeto do negócio (identidade persistente) em duas sessões diferentes, as duas instâncias serão realmente "diferentes" (identidade de JVM). Os conflitos são resolvidos usando (versionamento automático) no flush/commit, usando abordagem otimista.

Este caminho deixa o Hibernate e o banco dedados se preocuparem com a concorrência; também fornece uma escalabilidade melhor, garantindo que a identidade em unidades de trabalho único-encadeadas não necessite de bloqueio dispendioso ou de outros meios de sincronização. A aplicação nunca necessita sincronizar qualquer objeto de negócio tão longo que transpasse uma única thread por `Session`. Dentro de uma `Session` a aplicação pode usar com segurança o `==` para comparar objetos.

Com tudo, uma aplicação que usa `==` fora de uma `Session`, pode ver resultados inesperados. Isto pode ocorrer mesmo em alguns lugares inesperados, por exemplo, se você colocar duas instâncias desacopladas em um mesmo `Set`. Ambos podem ter a mesma identidade na base de dados (isto é eles representam a mesma linha em uma tabela), mas a identidade da JVM pela definição não garantida para instâncias em estado desacoplado. O

desenvolvedor tem que sobrescrever os métodos `equals()` e `hashCode()` em classes persistentes e implementar sua própria noção da igualdade do objeto. Advertência: nunca use o identificador da base de dados para implementar a igualdade, use atributos de negócio, uma combinação única, geralmente imutável. O identificador da base de dados mudará se um objeto transiente passar para o estado persistente. Se a instância transiente (geralmente junto com instâncias desacopladas) for inserida em um `Set`, mudar o hashcode quebra o contrato do `Set`. Atributos para chaves de negócio não têm que ser tão estável quanto às chaves primárias da base de dados, você somente tem que garantir a estabilidade durante o tempo que os objetos estiverem no mesmo Set. Veja o website do Hibernate para uma discussão mais completa sobre o assunto. Note também que esta não é uma caracteristica do Hibernate, mas simplesmente como a identidade e a igualdade do objeto de Java têm que ser implementadas.

11.1.4. Edições comuns

Nunca use o anti-patterns *session-per-user-session* ou *session-per-application* (naturalmente, há umas exceções raras a essa regra). Note que algumas das seguintes edições podem também aparecer com patterns recomendados, certifique-se que tenha compreendido as implicações antes de fazer uma decisão de projeto:

- Uma `Session` não é threadsafe. As coisas que são supostas para trabalhar concorrentemente, como requisições HTTP, session beans, ou Swing, causarão condições de disputa se uma instância `Session` for compartilhada. Se você mantiver sua `Session` do Hibernate em seu `HttpSession` (discutido mais tarde), você deve considerar sincronizar o acesso a sua sessão do HTTP. Caso contrário, um usuário que clica em reload várias muito rapidamente pode usar o mesmo `Session` em duas threads executando concorrentemente.

- Uma exceção lançada pelo Hibernate significa que você tem que dar rollback na sua transação no banco de dados e fechar a `Session` imediatamente (discutido mais tarde em maiores detalhes). Se sua `Session` é limitado pela aplicação, você tem que parar a aplicação. Dando rollback na transação no banco de dados não põe seus objetos do negócio em um estado anterior que estavam no início da transação. Isto significa que o estado da base de dados e os objetos de negócio perdem a sincronização. Geralmente não é um problema porque as exceções não são recuperáveis e você tem que iniciar após o rollback de qualquer maneira.

- O `Session` guarda em cache cada objeto que está no estado persistente (guardado e checado para estado "sujo" pelo Hibernate). Isto significa que ele cresce infinitamente até que você obtenha uma OutOfMemoryException, se você o mantiver aberto por muito tempo ou simplesmente carregar dados demais. Uma solução é chamar `clear()` e `evict()` para controlar o cache da `Session`, mas você deve considerar uma Store Procedure se precisar de operações que envolvam grande volume de dados. Algumas soluções são mostradas no Capítulo 13, *Processamento de lotes*. Manter uma `Session` aberta durante uma sessão do usuário significa também uma probabilidade elevada de se acabar com dados velhos.

11.2. Demarcação de transações de bancos de dados

Os limites de uma transação de banco de dados (ou sistema) são sempre necessários. Nenhuma comunicação com o banco de dados pode ocorrer fora de uma transação de banco de dados (isto parece confundir muitos desenvolvedores que estão usados modo auto-commit). Sempre use os limites desobstruídos da transação, até mesmo para operações somente leitura. Dependendo de seu nível de isolamento e capacidade da base de dados isto pode não ser requerido, mas não há nenhum aspecto negativo se você demarca sempre transações explicitamente. Certamente, uma única transação será melhor executada do que muitas transações pequenas, até mesmo para dados de leitura.

Uma aplicação do Hibernate pode funcionar em ambientes não gerenciados (isto é aplicações standalone, Web simples ou Swing) e ambientes gerenciados J2EE. Em um ambiente não gerenciado, o Hibernate é geralmente responsável pelo seu próprio pool de conexões. O desenvolvedor tem que manualmente ajustar limites das tran-

sações, ou seja, começar, commitar, ou dar rollback nas transações ele mesmo. Um ambiente gerenciado fornece transações gerenciadas por contêiner (CMT - container-managed transactions), com um conjunto da transações definido declarativamente em descritores de deployment de EJB session beans, por exemplo. A demarcação programática é então já não é necessário.

Entretanto, é freqüentemente desejável manter sua camada de persistência portável entre ambientes de recurso locais não gerenciados e sistemas que podem confiar em JTA, mas usar BMT em vez de CMT. Em ambos os casos você usaria demarcação de transação programática. O Hibernate oferece uma API chamada Transaction que traduz dentro do sistema de transação nativa de seu ambiente de deployment. Esta API é realmente opcional, mas nós encorajamos fortemente seu uso a menos que você estiver em um CMT session bean.

Geralmente, finalizar um `Session`envolve quatro fases distintas:

- flush da sessão
- commitar a transação
- fechar a sessão
- tratar as exceções

A limpeza da sessão já foi bem discutida, agora nós daremos uma olhada na demarcação da transação e na manipulação de exceção em ambientes controlados e não controlados.

11.2.1. Ambiente não gerenciado

Se uma camada de persistência do Hibernate roda em um ambiente não gerenciado, as conexões do banco de dados são geralmente tratadas pelos pools de conexões simples (isto é, não baseados em DataSource) dos quais o Hibernate obtém as conexões assim que necessita. A maneira de se manipular uma sessão/transação é mais ou menos assim:

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Você não pode chamar `flush()` do `Session()` explicitamente - a chamada ao `commit()` dispara automaticamente a sincronização para a sessão (dependendo do Seção 10.10, “Limpando a Session”). Uma chamada ao `close()` marca o fim de uma sessão. A principal implicação do `close()` é que a conexão JDBC será abandonada pela sessão. Este código Java é portável e funciona em ambientes não gerenciado e de JTA.

Uma solução muito mais flexível é gerência integrada de contexto da "sessão atual" do Hibernate, como descrito anteriormente:

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();
```

```
    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

Você muito provavelmente nunca verá estes fragmentos de código em uma aplicação regular; as exceções fatais (do sistema) devem sempre ser pegas no "alto". Ou seja, o código que executa chamadas do Hibernate (na camada de persistência) e o código que trata `RuntimeException` (e geralmente pode somente limpar acima e na saída) estão em camadas diferentes. O gerenciamento do contexto atual feito pelo Hibernate pode significativamente simplificar este projeto, como tudo que você necessita é do acesso a um `SessionFactory`. A manipulação de exceção é discutida mais tarde neste capítulo.

Note que você deve selecionar `org.hibernate.transaction.JDBCTransactionFactory` (que é o padrão) e para o segundo exemplo `"thread"` como seu `hibernate.current_session_context_class`.

11.2.2. Usando JTA

Se sua camada de persistência funcionar em um servidor de aplicação (por exemplo, dentro dos EJB session beans), cada conexão do datasource obtida pelo Hibernate automaticamente fará parte da transação global de JTA. Você pode também instalar uma implementação standalone de JTA e usá-la sem EJB. O Hibernate oferece duas estratégias para a integração de JTA.

Se você usar bean-managed transactions (BMT - transações gerenciadas por bean) o Hibernate dirá ao servidor de aplicação para começar e para terminar uma transação de BMT se você usar a API Transaction. Assim, o código de gerência de transação é idêntico ao ambiente não gerenciado.

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Se você quiser usar um `Session` limitada por transação, isto é, a funcionalidade do `getCurrentSession()` para a propagação fácil do contexto, você terá que usar diretamente a API JTA `UserTransaction`:

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                            .lookup("java:comp/UserTransaction");

    tx.begin();

    // Do some work on Session bound to transaction
```

```
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

Com CMT, a demarcação da transação é feita em descritores de deployment do session beans, não programaticamente, conseqüentemente, o código é reduzido a:

```
// CMT idiom
 Session sess = factory.getCurrentSession();

 // do some work
 ...
```

Em um CMT/EJB mesmo um rollback acontece automaticamente, desde que uma exeção `RuntimeException` não tratável seja lançada por um método de um session bean que informa ao contêiner ajustar a transação global ao rollback. *Isto significa que você não necessita usar a API `Transaction` do Hibernate em tudo com BMT ou CMT e você obtém a propagação automática do Session "atual" limitada à transação.*

Veja que você deverá escolher `org.hibernate.transaction.JTATransactionFactory` se você usar o JTA diretamente (BMT) e `org.hibernate.transaction.CMTTransactionFactory` em um CMT session bean, quando você configura a fábrica de transação do Hibernate. Lembre-se também de configurar o `hibernate.transaction.manager_lookup_class`. Além disso, certifique-se que seu `hibernate.current_session_context_class` ou não é configurado (compatibilidade com o legado) ou é definido para `"jta"`.

A operação `getCurrentSession()` tem um aspecto negativo em um ambiente JTA. Há uma advertência para o uso do método liberado de conexão `after_statement`, o qual é usado então por padrão. Devido a uma limitação simples da especificação JTA, não é possível para o Hibernate automaticamente limpar quaisquer instâncias `ScrollableResults` ou `Iterator` abertas retornadas pelo `scroll()` ou `iterate()`. Você *deve* liberar o cursor subjacente da base de dados chamando `ScrollableResults.close()` ou `Hibernate.close(Iterator)` explicitamente de um bloco `finally`. (Claro que a maioria de aplicações podem facilmente evitar o uso do `scroll()` ou do `iterate()` em todo código provindo do JTA ou do CMT.)

11.2.3. Tratamento de Exceção

Se a `Session` levantar uma exceção (incluindo qualquer `SQLException`), você deve imediatamente dar um rollback na transação do banco, chamando `Session.close()` e descartando a instância da `Session`. Certos métodos da `Session` não deixarão a sessão em um estado inconsistente. Nenhuma exceção lançada pelo Hibernate pode ser recuperada. Certifique-se que a `Session` será fechada chamando `close()` no bloco `finally`.

A exceção `HibernateException`, a qual envolve a maioria dos erros que podem ocorrer em uma camada de persistência do Hibernate, é uma exceção unchecked (não estava em umas versões mais antigas do Hibernate). Em nossa opinião, nós não devemos forçar o desenvolvedor a tratar uma exceção irrecuperável em uma camada mais baixa. Na maioria dos sistemas, as exceções unchecked e fatais são tratadas em um dos primeiros frames da pilha da chamada do método (isto é, em umas camadas mais elevadas) e uma mensagem de erro é apresentada ao usuário da aplicação (ou a alguma outra ação apropriada é feita). Note que Hibernate pode também lançar outras exceções unchecked que não são um `HibernateException`. Estas, também são, irrecuperáveis e uma ação apropriada deve ser tomada.

O Hibernate envolve `SQLException`s lançadas ao interagir com o banco de dados em um `JDBCException`. Na realidade, o Hibernate tentará converter a exceção em em uma sub classe mais significativa da `JDBCException`. A `SQLException` subjacente está sempre disponível através de `JDBCException.getCause()`.

O Hibernate converte a `SQLException` em uma sub classe `JDBCException` apropriada usando `SQLExceptionConverter` associado ao SessionFactory. Por padrão, o `SQLExceptionConverter` é definido pelo dialeto configurado; entretanto, é também possível conectar em uma implementação customizada (veja o javadoc para mais detalhes da classe `SQLExceptionConverterFactory`). Os subtipos padrão de `JDBCException` são:

- `JDBCConnectionException` - indica um erro com a comunicação subjacente de JDBC.
- `SQLGrammarException` - indica um problema da gramática ou da sintaxe com o SQL emitido.
- `ConstraintViolationException` - indica algum forma de violação de confinamento de integridade.
- `LockAcquisitionException` - indica um erro ao adquirir um nível de bloqueio necessário para realizar a operação de requisição.
- `GenericJDBCException` - uma exceção genérica que não cai em algumas das outras categorias.

11.2.4. Timeout de Transação

Uma característica extremamente importante fornecida por um ambiente gerenciado como EJB e que nunca é fornecido pelo código não gerenciado é o timeout de transação. Timeouts de transação asseguram que nenhuma transação possa reter indefinidamente recursos enquanto não retorna nenhuma resposta ao usuário. Fora de um ambiente controlado (JTA), o Hibernate não pode fornecer inteiramente esta funcionalidade. Entretanto, o Hibernate pode afinal controlar as operações do acesso a dados, assegurando que o nível de deadlocks e queries do banco de dados com imensos resultados definidos sejam limitados pelo timeout. Em um ambiente gerenciado, o Hibernate pode delegar o timeout da transação ao JTA. Esta funcionalidade é abstraída pelo objeto `Transaction` do Hibernate.

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();

    // do some work
    ...

    sess.getTransaction().commit()
}
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Veja que `setTimeout()` não pode ser chamado em um CMT bean, onde os timeouts das transações devem ser definidos declarativamente.

11.3. Controle de concorrência otimista

O único caminho que é consistente com a elevada concorrência e escalabilidade é controle de concorrência otimista com versionamento. Checagem de versão usa número de versão, ou timestamps, para detectar conflitos de atualizações (e para impedir atualizações perdidas). O Hibernate fornece três caminhos possíveis para escrever aplicações que usam concorrência otimista. Os casos de uso que nós mostramos estão no contexto de con-

versações longas, mas a checagem de versão também tem o benefício de impedir atualizações perdidas em únicas transações.

11.3.1. Checagem de versão da aplicação

Em uma implementação sem muita ajuda do Hibernate, cada interação com o banco de dados ocorre em uma nova `Session` e o desenvolvedor é responsável para recarregar todas as instâncias persistentes da base de dados antes de manipulá-las. Este caminho força a aplicação a realizar sua própria checagem de versão para assegurar a conversação do isolamento da transação. Este caminho é menos eficiente em termos de acesso ao banco de dados. É a caminho mais similar a EJBs entity.

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();

int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");

t.commit();
session.close();
```

A propriedade `version` é mapeada usando `<version>`, e o Hibernate vai incrementá-lo-á automaticamente durante o flush se a entidade estiver alterada.

Claro, se você se estiver operando em um ambiente de baixa concorrência de dados e não requerer a checagem de versão, você pode usar este caminho e apenas saltar a checagem de versão. Nesse caso, o *ultimo commit realizdo* é a estratégia padrão para suas conversações longas. Mantenha em mente que isto pode confundir os usuários da aplicação, assim como eles podem experimentar atualizações perdidas sem mensagens de erro ou uma possibilidade ajustar mudanças de conflito.

Claro que, checagem manual da versão é somente praticável em circunstâncias triviais e não para a maioria de aplicações. Freqüentemente, os grafos completos de objetos modificados têm que ser verificados, não somente únicas instâncias. O Hibernate oferece checagem de versão automática com uma `Session` estendida ou instâncias desatachadas como o paradigma do projeto.

11.3.2. Sessão estendida e versionamento automático

Uma única instância de `Session` e suas instâncias persistentes são usadas para a conversação inteira, isto é conhecido como *session-per-conversation*. O Hibernate verifica versões da instância no momento dio flush, lançando uma exceção se a modificação concorrente for detectada. Até o desenvolvedor pegar e tratar essa exceção (as opções comuns são a oportunidade para que o usuário intercale as mudanças ou reinicie a conversação do negócio com dados não antigos).

A `Session` é desconectada de toda a conexão JDBC subjacente enquanto espera a interação do usuário. Este caminho é a mais eficiente em termos de acesso a bancos de dados. A aplicação não necessita concernir-se com a checagem de versão ou com as instâncias destacadas reatadas, nem tem que recarregar instâncias em cada transação.

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC connection, start transaction

foo.setProperty("bar");

session.flush();    // Only for last transaction in conversation
```

```
t.commit();         // Also return JDBC connection
session.close();    // Only for last transaction in conversation
```

O objeto `foo` sabe que `Session` já foi carregada. Começando uma nova transação em uma sessão velha obtém uma conexão nova e recomeça a sessão. Commitando uma transação desconecta uma sessão da conexão JDBC e retorna a conexão ao pool. Após a reconexão, forçar uma checagem de versão em dados que você não está atualizando, você pode chamar `Session.lock()` com o `LockMode.READ` em todos os objetos que possam ter sido atualizados por uma outra transação. Você não necessita bloquear nenhum dado para atualizar. Geralmente você configuraria `FlushMode.NEVER` em uma `Session` estendida, de modo que somente o último ciclo da transação tenha permissão de persistir todas as modificações feitas nesta conversação. Disso, somente esta última transação incluiria a operação `flush()` e então chamar também `close()` da sessão para terminar a conversação.

Este pattern é problemático se a `Session` for demasiadamente grande para ser armazenado durante o tempo que usuário pensar, por exemplo um `HttpSession` estiver mantido tão pequeno quanto possível. Como o `Session` é também cache de primeiro nível (imperativo) e contém todos os objetos carregados, nós podemos provavelmente usar esta estratégia somente para alguns ciclos de requisição/resposta. Você deve usar a `Session` somente para uma única conversação, porque ela logo também estará com dados velhos.

(Note que versões mais atuais do Hibernate requerem a desconexão e o reconexão explícitas de uma `Session`. Estes métodos são desatualizados, como o início e término de uma transação tem o mesmo efeito.)

Note também que você deve manter a `Session` desconectada fechada para a camada de persistência. Ou seja, use um EJB stateful session bean para prender a `Session` em um ambiente do três camadas e não o transferir à camada web (ou até serializá-lo para uma camada separada) para armazená-lo no `HttpSession`.

O pattern sessão estendida, ou *session-per-conversation*, é mais difícil de implementar com gerenciamento automático de sessão atual. Você precisa fornecer sua própria implementação do `CurrentSessionContext` para isto (veja o Hibernate Wiki para exemplos).

11.3.3. Objetos destacados e versionamento automático

Cada interação com o armazenamento persistente ocorre em uma `Session` nova. Entretanto, as mesmas instâncias persistentes são reusadas para cada interação com o banco de dados. A aplicação manipula o estado das instâncias desatachadas originalmente carregadas em um outro `Session` e reata-os então usando `Session.update()`, `Session.saveOrUpdate()` ou `Session.merge()`.

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been loaded already
t.commit();
session.close();
```

Outra vez, o Hibernate verificará versões da instância durante o flush, lançando uma exceção se ocorrer conflitos de atualizações.

Você pode também chamar o `lock()` em vez de `update()` e usar `LockMode.READ` (executando uma checagem de versão, ignorando todos os caches) se você estiver certo de que o objeto não foi modificado.

11.3.4. Versionamento automático customizado

Você pode desabilitar o incremento da versão automática do Hibernate para propriedades e coleções particulares configurando o mapeamento do atributo `optimistic-lock` para false. O Hibernate então não irá incrementa

versões se a propriedade estiver modificada.

Os esquemas da base de dados legada são freqüentemente estáticos e não podem ser modificados. Ou outras aplicações puderam também acessar a mesma base de dados e não sabem tratar a versão dos números ou timestamps. Em ambos os casos, o versionamento não pode confiar em uma coluna particular em uma tabela. Para forçar uma checagem de versão sem uma versão ou mapeamento da propriedade do timestamp com uma comparação do estado de todos os campos em uma linha, configure `optimistic-lock="all"` no mapeamento `<class>`. Note que isto conceitualmente é somente feito em trabalhos se Hibernate puder comparar o estado velho e novo, isto é, se você usa um único `Session` longo e não session-per-request-with-detached-objects.

Às vezes a modificação concorrente pode ser permitida tão longa quanto às mudanças que tiveram sido feitas que não sobrepuseram. Se você configurar `optimistic-lock="dirty"` ao mapear o `<class>`, o Hibernate comparará somente campos modificados durante o flush.

Em ambos os casos, com as colunas dedicadas da versão/timestamp ou com comparação do campo cheio/ modificados, o Hibernate usa uma única declaração UPDATE (com uma cláusula WHERE apropriada) por entidade para executar a checagem da versão e atualizar a informação. Se você usa a persistência transitiva para cascatear o reatamento das entidades associadas, o Hibernate pode executar atualizações desnecessárias. Isso não é geralmente um problema, mas triggers *on update* em um banco de dados podem ser executados mesmo quando nenhuma mudança foi feita nas instâncias destacadas. Você pode customizar este comportamento configurando `select-before-update="true"` no mapeamento `<class>`, forçando o Hibernate a dá um SELECT nas instâncias para assegurar-se esse as mudanças ocorreram realmente, antes de atualizar a linha.

11.4. Locking pessimista

Não se pretende que os usuários gastem muitas horas se preocupando com suas estratégias de locking. Geralmente é o bastante especificar um nível de isolamento para as conexões JDBC e então simplesmente deixar o banco de dados fazer todo o trabalho. Entretanto, os usuários avançados podem às vezes desejar obter locks pessimistas exclusivos, ou re-obter locks no início de uma nova transação.

O Hibernate usará sempre o mecanismo de lock da base de dados, nunca trava objetos na memória!

A classe `LockMode` define os diferentes níveis de lock que o Hibernate pode adquirir. Um lock é obtido pelos seguintes mecanismos:

- `LockMode.WRITE` é adquirido automaticamente quando o Hibernate atualiza ou insere uma linha.
- `LockMode.UPGRADE` pode ser adquirido explicitamente pelo usuário usando `SELECT ... FOR UPDATE` em um banco de dados que suporte esse sintaxe.
- `LockMode.UPGRADE_NOWAIT` pode ser adquirido explicitamente pelo usuário usando `SELECT ... FOR UPDATE NOWAIT` no Oracle.
- `LockMode.READ` é adquirido automaticamente quando o Hibernate lê dados em um nível Repeatable Read ou Serializable isolation. Pode ser readquirido explicitamente pelo usuário.
- `LockMode.NONE` representa a ausência do lock. Todos os objetos mudam para esse estado de lock no final da `Transaction`. Objetos associados com a sessão através do método `update()` ou `saveOrUpdate()` também são inicializados com esse lock mode.

O lock obtido "explicitamente pelo usuário" se dá em uma das seguintes maneiras:

- Uma chamada a `Session.load()`, especificando o `LockMode`.
- Uma chamada a `Session.lock()`.
- Uma chamada a `Query.setLockMode()`.

Se uma `Session.load()` é invocada com `UPGRADE` ou `UPGRADE_NOWAIT`, e o objeto requisitado ainda não foi car-

regado pela sessão, o objeto é carregado usando `SELECT ... FOR UPDATE`. Se `load()` for chamado para um objeto que já foi carregado com um lock menos restritivo que o novo lock solicitado, o Hibernate invoca o método `lock()` para aquele objeto.

O método `Session.lock()` executa uma verificação no número da versão se o modo de lock especificado for `READ`, `UPGRADE` ou `UPGRADE_NOWAIT`.. (No caso do `UPGRADE` ou `UPGRADE_NOWAIT`, é usado `SELECT ... FOR UPDATE`.)

Se o banco de dados não suportar o lock mode solicitado, o Hibernate vai usar um modo alternativo apropriado (ao invés de lançar uma exceção). Isso garante que a aplicação vai ser portável.

11.5. Modos de liberar a Connection

O comportamento legado do Hibernate (2.x) em consideração ao gerenciamento da conexão via JDBC fez com que a `Session` precisasse obter uma conexão quando ela precisasse pela primeira vez e depois manter a conexão enquanto a sessão não fosse fechada. O Hibernate 3.x introduz a idéia de modos de liberar a sessão, para informar a sessão a forma como deve manusear a sua conexão JDBC. Veja que essa discussão só é pertinente para conexões fornecidas com um `ConnectionProvider` configurado; conexões fornecidas pelo usuário estão fora do escopo dessa discussão. Os diferentes modos de liberação estão definidos pelos valores da enumeração `org.hibernate.ConnectionReleaseMode`:

- `ON_CLOSE` - essencialmente é o modo legado descrito acima. A sessão do Hibernate obtêm a conexão quando precisar executar alguma operação JDBC pela primeira vez e mantem enquanto a conexão não for fechada.
- `AFTER_TRANSACTION` – informa que a conexão deve ser liberada após a conclusão de uma `org.hibernate.Transaction`.
- `AFTER_STATEMENT` (também conhecida com liberação agressiva) – informa que a conexão deve ser liberada após a execução de cada statement. A liberação agressiva não ocorre se o statement deixa pra trás algum recurso aberto associado com a sessão obtida; atualmente, a única situação em que isso é possível é com o uso de `org.hibernate.ScrollableResults`.

O parâmetro de configuração `hibernate.connection.release_mode` é usado para especificar qual modo de liberação deve ser usado. Opções disponíveis:

- `auto` (padrão) – essa opção delega ao modo de liberação retornado pelo método `org.hibernate.transaction.TransactionFactory.getDefaultReleaseMode()`. Para JTATransactionFactory, ele retorna ConnectionReleaseMode.AFTER_STATEMENT; para JDBCTransactionFactory, ele retorna ConnectionReleaseMode.AFTER_TRANSACTION. Raramente é uma boa idéia alterar padrão, como frequencia ao se fazer isso temos falhas que parecem bugs e/ou suposições inválidas no código do usuário.
- `on_close` - indica o uso da ConnectionReleaseMode.ON_CLOSE. Essa opção foi deixada para manter a compatibilidade, mas seu uso é fortemente desencorajado.
- `after_transaction` – indica o uso da ConnectionReleaseMode.AFTER_TRANSACTION. Essa opção nada deve ser usada com ambientes JTA. Também note que no caso da ConnectionReleaseMode.AFTER_TRANSACTION, se a sessão foi colocada no modo auto-commit a conexão vai ser liberada de forma similar ao modo AFTER_STATEMENT.
- `after_statement` – indica o uso ConnectionReleaseMode.AFTER_STATEMENT. Adicionalmente, o `ConnectionProvider` configurado é consultado para verificar se suporta essa configuração ((`supportsAggressiveRelease()`). Se não suportar, o modo de liberação é redefinido como ConnectionRelease-Mode.AFTER_TRANSACTION. Essa configuração só é segura em ambientes onde podemos readquirir a mesma conexão JDBC toda vez que o método `ConnectionProvider.getConnection()` for chamado ou em um ambiente auto-commit onde não importa se nós recuperamos a mesma conexão.

Capítulo 12. Interceptadores e Eventos

É muito útil quando a aplicação precisa executar alguma "coisa" no momento em que o Hibernate executa uma de suas ações. Isso permite a implementação de certas funções genéricas, assim como permite estender as funcionalidades do Hibernate

12.1. Interceptadores

A interface `Interceptor` permite fornecer informações da session para o aplicativo, permitindo ao aplicativo inspecionar e/ou manipular as propriedades de um objeto persistente antes de ser salvo, atualizado, excluído ou salvo. Um dos possíveis usos é gerar informações de auditoria. Por exemplo, o seguinte `Interceptor` seta automaticamente o atributo `createTimestamp` quando um `Auditable` é criada e atualiza o atributo `lastUpdateTimestamp` quando um `Auditable` é atualizado.

Você pode implementar `Auditable` diretamente ou pode estender `EmptyInterceptor`, sendo que a segunda é considerada a melhor opção.

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
```

```
                        String[] propertyNames,
                        Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
                        Serializable id,
                        Object[] state,
                        String[] propertyNames,
                        Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
            System.out.println("Creations: " + creates + ", Updates: " + updates, "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

Os interceptadores podem ser aplicados em dois diferentes escopos: No escopo da `Session` e no escopo `SessionFactory`.

Um interceptador no escopo da `Session` é definido quando uma sessão é aberta usando o método sobrecarregado da SessionFactory.openSession() que aceita um `Interceptor` como parâmetro.

```
Session session = sf.openSession( new AuditInterceptor() );
```

Um interceptador no escopo da `SessionFactory` é definido no objeto `Configuration` antes da `SessionFactory` ser instanciada. Nesse caso, o interceptador fornecido será aplicado para todas as sessões abertas por aquela `SessionFactory`; Isso apenas não ocorrerá caso seja especificado um interceptador no momento em que a sessão for aberta. Um interceptador no escopo de `SessionFactory` deve ser thread safe, tomando-se o cuidado de não armazenar atributos de estado específicos da sessão, pois, provavelmente, múltiplas sessões irão utilizar esse interceptador simultaneamente.

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

12.2. Sistema de Eventos

Se você precisa executar uma ação em determinados eventos da camada de persistência, você também pode usar a arquitetura de *event* do Hibernate3. Um evento do sistema pode ser utilizado como complemento ou em substituição a um interceptador.

Essencialmente todos os métodos da interface `Session` possuem um evento correlacionado. Se você tiver um

`LoadEvent`, um `LoadEvent`, etc (consulte o DTD do XML de configuração ou o pacote `org.hibernate.event` para a lista completa dos tipos de eventos). Quando uma requisição é feita em um desses métodos, a `Session` do hibernate gera um evento apropriado e o envia para o listener de evento correspondente àquele tipo de evento. Esses listeners implementam a mesma lógica que aqueles métodos, trazendo os mesmos resultados. Entretanto, você é livre para implementar uma customização de um desses listeners (isto é, o `LoadEvent` é processado pela implementação registrada da interface `LoadEventListener`), então sua implementação vai ficar responsável por processar qualquer requisição `load()` feita pela `Session`.

Para todos os efeitos esses listeners deve ser considerados singletons; ou seja, eles são compartilhados entre as requisições, e assim sendo, não devem salvar nenhum estado das variáveis instanciadas.

Um listener personalizado deve implementar a interface referente ao evento a ser processado e/ou deve estender a classes base equivalente (ou mesmo os listeners padrões usados pelo Hibernate, eles não são declarados como finais com esse objetivo). O listener personalizado pode ser registrado programaticamente no objeto `Configuration`, ou declarativamente no XML de configuração do Hibernate (o registro do listener no propertie de configuração não é suportado). Aqui temos um exemplo de como carregar um listener personalizado:

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListener interface
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
            throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

Você também precisa adicionar uma entrada no XML de configuração do Hibernate para registrar declarativamente qual listener deve se utilizado em conjunto com o listener padrão:

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
            <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

Ou, você pode registrar o listener programaticamente:

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new DefaultLoadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

Listeners registrados declarativamente não compartilham da mesma instancia. Se o mesmo nome da classe é utilizado em vários elementos e `<listener/>`, cada um vai resultar em uma instancia separada dessa classe. Se você tem a necessidade de compartilhar uma instancia de um listener entre diversos tipos de listeners você deve registrar o listener programaticamente.

Mas porque implementar uma interface e definir o tipo específico durante a configuração? Bem, um listener pode implementar vários listeners de evento. Com o tipo sendo definido durante o registro, fica fácil ligar ou desligar listeners personalizados durante a configuração.

12.3. Segurança declarativa no Hibernate

Normalmente, a segurança declarativa em aplicações Hibernate é administrada na camada session facade. Agora, o Hibernate3 permite sejam dadas permissões a certas ações vai JACC, e aturorização via JAAS. Esta é funcionalidade opcional construída em cima da arquitetura de evento.

Primeiro, você tem que configurar os eventos listeners apropriados, para habilitar o uso de autorização de JAAS.

```
<listener type="pre-delete" class="org.hibernate.secure.JACCPreDeleteEventListener"/>
<listener type="pre-update" class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert" class="org.hibernate.secure.JACCPreInsertEventListener"/>
<listener type="pre-load" class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

Veja que `<listener type="..." class="..."/>` é um atalho para `<event type="..."><listener class="..."/></event>` quando existe apenas um listener um tipo particular de evento .

Agora, em `hibernate.cfg.xml`, associe as permissões a papéis:

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions="*"/>
```

Os nomes dos papíes e os papéis são entendidos por seu provedor JACC.

Capítulo 13. Processamento de lotes

Uma alternativa para inserir 100.000 linhas no banco de dados usando o Hibernate pode ser a seguinte:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

Isto irá falhar em algum lugar próximo a linha 50.000, lançando uma `OutOfMemoryException`. Isso ocorre devido ao fato do Hibernate fazer cache de todas as instâncias de `Customer` inseridas num cachê em nível de sessão.

Neste capítulo veremos como contornar esse problema. Entretanto, se você vai realizar processamento de lotes, é muito importante que você habilite o uso de lotes JDBC, se você pretende obter um desempenho razoável. Defina o tamanho do lote JDBC em um valor razoável (algo entre 10-50):

```
hibernate.jdbc.batch_size 20
```

Veja que o Hibernate desabilita o insert em lote a nível JDBC de forma transparente se você usa o gerador de identificadore do tipo `identiy`

Você também pode querer rodar esse tipo de processamento de lotes com o cache secundário completamente desabilitado:

```
hibernate.cache.use_second_level_cache false
```

Mas isto não é absolutamente necessário, desde que nós possamos ajustar o `CacheMode` para desabilitar a interação com o cache secundário.

13.1. Inserção de lotes

Quando você estiver inserindo novos objetos persistentes, vocês deve executar os métodos `flush()` e `clear()` regularmente na sessão, para controlar o tamanho do cache primário.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.2. Batch updates

Para recuperar e atualizar informações a mesma idéia é válida. Adicionalmente, pode precisar usar o `scroll()` para usar recursos no lado do servidor em queries que retornam muita informação.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.3. A interface StatelessSession

Alternativamente, o Hibernate provê uma API orientada à comandos, usada para transmitir um fluxo de dados de e para o banco de dados na forma de objetos soltos. Uma `StatelessSession` não tem um contexto persistente associado e não fornece muito das semânticas de alto nível para controle do ciclo de vida. Em especial, uma StatelessSession não implemente o cache primário e nem interage com o cache secundário ou query cache. Ele não implementa salvamento transacional automatico ou checagem automática de mudanças. Operação realizadas usando uma StatelessSession não fazem nenhum tipo de cascade com as instancias associadas. As coleções são ignoradas por uma StatelessSession. Operações realizadas com um StatelessSession ignoram a arquitetura de eventos e os interceptadores. StatelessSession são vulneráveis aos efeitos do aliasing dos dados, devido a falta do cache primário. Uma StatelessSession é uma abstração de baixo nível, muito mais próxima do JDBC.

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

Veja neste exempo, as instancias de `Customer` retornadas pela query são imediatamente desvinculadas. Elas nunca serão assossiadas à um contexto persistente.

As operações `insert(), update()` e `delete()` definidos pela interface `StatelessSession` são considerados operações diretas no banco de dados (row-level operations), isso resulta em uma execução imediata de comandos SQL `INSERT, UPDATE` ou `DELETE` respectivamente. Devido a isso, eles possuem uma semântica bem diferente das operações `save(), saveOrUpdate()` ou `delete()` definidas na interface `Session`.

13.4. Operações no estilo DML

Como já discutido, mapeamento objeto/relacional automático e transparente é conseguido com a gerência do estado do objeto. Com isto o estado daquele objeto fica disponível na memória, manipulando(usando as expressões SQL `Data Manipulation Language` (SQL-style DML): `INSERT`, `UPDATE`, `DELETE`) os dados diretamente no banco de dados não irá afetar o estado registrado em memória. Entretanto, o Hibernate provê métodos para executar queries SQL-style DML, que são totalmente executas com HQL (Hibernate Query Language) (Capítulo 14, *HQL: A linguagem de Queries do Hibernate*).

A pseudo-sintaxe para expressões `UPDATE` e `DELETE` é: `( UPDATE | DELETE ) FROM? NomeEntidade (WHERE condições_where)?`. Algumas observações:

- Na clausula from, a palavra chave FROM é opcional;
- Somente uma entidade pode ser chamada na clausula from; opcionalmente pode ter um alias. Se o nome da entidade for possuir um alias, então qualquer propriedade referenciada deve usar esse alias qualificado; se o nome da entidade não possuir um alias, então nenhuma das propriedade precisa usar o acesso qualificado.
- Na Seção 14.4, “Formas e sintaxe de joins” (ambas implícita ou explicita) pode ser especificada em um bulk HQL query. Sub-queries podem ser usadas na clausula where; as subqueries podem conter joins.
- A clausula where também é opcional.

Como exemplo para executar um HQL `UPDATE`, use o método `Query.executeUpdate()`(o método ganhou o nome devido a sua familiaridade com o do JDBC `PreparedStatement.executeUpdate()`):

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
// or String hqlUpdate = "update Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Por default, declarações de `UPDATE` HQL, não efetam os valores das propriedades Seção 5.1.7, “version (opcional)” ou Seção 5.1.8, “timestamp (opcional)” para as entidades afetadas; isto está de acordo com a especificação EJB3. Porém, você pode forçar o Hibernate a reajustar os valores das propriedades `version` e `timestamp` através do uso de uma `atualização versionada`. Isso pode ser feito adicionando-se a palavra chave `VERSIONED` depois da palavra chave de `UPDATE`.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Veja que os tipos personalizados (`org.hibernate.usertype.UserVersionType`) não podem ser usados em conjunto com declarações de `atualização versionada`.

Para executar um HQL `DELETE`, use o mesmo método `Query.executeUpdate()`:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
```

```
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

O valor `int` retornado pelo método `Query.executeUpdate()` indica o numero de entidade afetadas pela operação. Lembre-se que isso pode estar ou não relacionado ao número de linhas alteradas no banco de dados. Uma operação bulk HQL pode resultar em várias expressões SQL reais a serem executadas, por exemplo, no caso de joined-subclass. O número retornado indica a quantidade real de entidades afetadas pela expressão. Voltando ao exemplo da joined-subclass, a exclusão de uma das subclasses pode resultar numa exclusão em outra tabelas, não apenas na tabela para qual a subclasses está mapeada, mas também tabela "root" e possivelmente nas tabelas joined-subclass num nível hierárquico imediatamente abaixo.

A pseudo-sintaxe para o comando `INSERT` é: `INSERT INTO EntityName properties_list select_statement`. Alguns pontos a observar:

- Apenas a forma INSERT INTO ... SELECT ... é suportada; INSERT INTO ... VALUES ... não é suportada.

  A lista de propriedade é análoga à `especificação da coluna` do comando SQL `INSERT`. Para entidades envolvidas em mapeamentos, apenas a propriedades definidas diretamente a nível da classe podem ser usandas na properties_list. Propriedades da superclass não são permitidas; e as propriedades da subclasse não faz sentido. Em outras palavras, os comandos `INSERT` não são polimorficos.
- O camando select pode ser qualquer query HQL válida, que tenha um retorno compatível com o tipo com o esperado pela inclusão. Atualmente, isto é verificado durante a compilação da query, isto é melhor do que permitir que a verificação chegue ao banco de dados. Entretanto perceba que isso pode causar problemas entre os `Tipo` do Hibernate que são *equivalentes* em oposição a *equal*. Isso pode causar problemas nas combinações entre a propriedade definida como `org.hibernate.type.DateTypee` um propriedade definida como `org.hibernate.type.TimestampType`, embora o banco de dados não possa fazer uma distinção ou possa ser capaz de manusear a conversão.
- Para a propriedade id, a expressão insert oferece duas opções. Você pode especificar qualquer propriedade id explicitamente no properties_list (em alguns casos esse valor é obtido diretamente da expressão select) ou pode omitir do properties_list (nesse caso, um valor gerado é usado). Essa ultima opção só é válida quando são usados geradores de ids que operam no banco de dados; a tentativa de usar essa opção com geradores do tipo "em memória" vai causar um exceção durante a etapa de parser. Veja a finalidades desta discussão, os seguintes geradores operam com o banco de dados `org.hibernate.id.SequenceGenerator` (e suas subclasses) e qualquer implementação de `org.hibernate.id.PostInsertIdentifierGenerator`. Aqui, a exceção mais notável é o `org.hibernate.id.TableHiLoGenerator`, que não pode ser usado porque ele não dispõe de mecanismos para recuperar o seu valor.
- Para propriedades mapeadas como `version` ou `timestamp`, a expressão insert oferece a você duas opções. Você pode especificar a propriedade na properties_list (nesse caso o seu valor é obtido a partir da expressão select correspondente) ou ele pode ser omitido da properties_list (neste caso o usa o `valor semente` definido pela classe `org.hibernate.type.VersionType`).

Exemplo da execução de um HQL `INSERT`:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c where
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

Capítulo 14. HQL: A linguagem de Queries do Hibernate

O Hibernate vem com uma poderosa linguagem que é (intencionalmente) muito parecida com o SQL. Mas não seja enganado pela sintaxe; a HQL é totalmente orientada à objetos, requer conhecimentos de herança, polimorfismo e associações.

14.1. Case Sensitíve

As consultas não são case-sensitive, exceto pelo nomes das classes e propriedades Java. `sELEct` e o mesmo que `SELECT` mas `org.hibernate.eg.FOO` não é `org.hibernate.eg.Foo` e `foo.barSet` não é `foo.BARSET`.

Esse manual usa as palavras chave HQL em letras minúsculas. Alguns usuários acham que com letras maiúsculas as queries ficam mais legíveis, mas nós achamos essa convenção feia dentro do código Java.

14.2. A clausula from

A mais simples query possível do Hibernate é a assim:

```
from eg.Cat
```

Ela irá retornar todas as instancias da classe `eg.Cat`. Necessariamente não precisamos qualificar o nome da classe, pois é realizado `auto-import` por padrão. Por isso na maior parte do tempos nós simplesmente escrevemos:

```
from Cat
```

Na maior parte do tempo, você precisará atribuir um *alias*, desde que você queira se referia ao `Cat` em outras partes da query.

```
from Cat as cat
```

Essa query atribui um alias a `cat` para as instancias de `Cat`, então nós podemos usar esse alias depois na query. A palavra chave as é opcional; poderíamos escrever assim:

```
from Cat cat
```

Múltiplas classes pode ser envolvidas, resultando em um produto cartesiano ou "cross" join.

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

É considerada uma boa prática os nomes dos aliases começarem com letra minúscula, aderente com os padrões Java para variáveis locais (ex: `domesticCat`).

14.3. Associações e joins

Nós também podemos querer atribuir aliases em uma entidade associada, ou mesmo em elementos de uma coleção de valores, usando um `join`.

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

Os tipos de joins suportados foram inspirados no SQL ANSI:

- `inner join`
- `left outer join`
- `right outer join`
- `full join` (geralmente não é útil)

As construções `inner join`, `left outer join` e `right outer join` podem ser abreviadas.

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

Você pode fornecer condições extras de join usando a palavra chave do HQL `with`.

```
from Cat as cat
    left join cat.kittens as kitten
        <literal>with</literal> kitten.bodyWeight > 10.0
```

Adicionalmente, um join do tipo "fetch" permite que associações ou coleções de valores sejam inicializadas junto com o objeto pai, usando apenas um select. Isso é muito útil no caso das coleções. Isso efetivamente sobrescreve as declarações outer join e lazy do arquivo mapeamento para associações e coleções. Veja a seção Seção 19.1, “Estratégias de Fetching” para mais informações.

```
from Cat as cat
    inner join <literal>fetch</literal>cat.mate
    left join <literal>fetch</literal>cat.kittens
```

Normalmmente, um join do tipo `fetch` não precisa um alias, pois o objeto associado não deve ser usado na clausula `where` (ou em qualquer outra clausula). Também, os objetos associados não são retornados diretamente nos resultados da query. Ao invés disso, eles devem ser acessados usando o objeto pai. A única razão que nós podemos necessitar de um alias é quando fazemos um fech join recursivamente em uma coleção adicional:

```
from Cat as cat
    inner join <literal>fetch</literal>cat.mate
    left join <literal>fetch</literal>cat.kittens child
    left join <literal>fetch</literal>child.kittens
```

Veja que a construção `fetch` não deve ser usada em queries invocadas usando `iterate()` (embora possa ser usado com `scroll()`). O `fetch` também não deve ser usado junto com o `setMaxResults()` ou `setFirstResult()` pois essas operações são baseadas nas linhas retornadas, que normalmente contem duplicidade devido ao fetching das coleções, então o número de linhas pode não ser o que você espera. O `fetch` não deve ser usado junto com uma condição ad hoc `with`. É possível que seja criado um produto cartesiano pelo join fetching em mais do que uma coleção em uma query, então tome cuidado nesses casos. Um join fetching em varias coleções pode trazer resultados inesperados para mapeamentos do tipo bag, tome cuidado na hora de formular queries

como essas. Finalmente, observe o seguinte, o `full join fetch` e `right join fetch` não são significativos.

Se você está usando a propriedade lazy (`com` instrumentação de bytecode), é possível forçar o Hibernate a `buscar` as propriedades lazy imediatamente (na primeira query), usando `fetch all properties` .

```
from Document <literal>fetch</literal>all properties order by name
```

```
from Document doc <literal>fetch</literal>all properties where lower(doc.name) like '%cats%'
```

14.4. Formas e sintaxe de joins

O HQL suporta duas formas de associação para união: `implícita` e `explicita`.

As queries apresentadas na seção anterior usam a forma `explicita`, onde a palavra chave "join" é explicitamente usada na clausula "from". Essa é a forma recomendada.

A forma `implícita` não usa a palavra chave "join". Entretanto, as associações são diferenciadas usando pontuação ("." - dotnation). Uniões implícitas podem aparecer em qualquer das clausulas HQL. A união `implícita` resulta em sentenças SQL que contem inner joins.

```
from Cat as cat where cat.mate.name like '%s%'
```

14.5. Clausula select

A clausula `select` seleciona quais objetos e propriedades retornam no resultado da query. Considere:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

A query selecionará `mates` (companheiros), de outros `Cats`. Atualmente, podemos expressar a query de forma mais compacta como:

```
select cat.mate from Cat cat
```

Queries podem retornar propriedades de qualquer tipo de valor, incluindo propriedades de tipo de componente:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

Queries podem retornar múltiplos objetos e/ou propriedades como um array do tipo Object[],

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

ou como um `List`,

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
```

```
    left outer join mother.kittens as offspr
```

ou como um objeto Java typesafe,

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

assumindo que a classe `Family` tenha um construtor apropriado.

Pode-se designar referencias a expressões selecionadas, `as`:

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

Isto é bem mais útil quando usado junto `com select new map`:

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

Esta query retorna um `Map` de referencias para valores selecionados.

14.6. Funções de agregação

As queries HQL podem retornar o resultado de funções agregadas nas propriedades.

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

As funções agregadas suportadas são:

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

Pode-se usar operadores aritiméticos, concatenação e funções SQL reconhecidas na clausula select:

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

As palavras `distinct` e `all` podem ser usadas e têm a mesma semântica como no SQL.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

14.7. Queries polimórficas

A query:

```
from Cat as cat
```

retorna instancias não só de `Cat`, mas também de subclasses como `DomesticCat`. As queries do Hibernate podem nomear qualquer classe Java ou interface na clausula `from`. A query retornará instancias de toda classe persistente que extenda a determinada classe ou implemente a determinada interface. A query , a seguir, pode retornar todo objeto persistente:

```
from java.lang.Object o
```

A interface `Named` pode ser implementada por várias classes persistentes:

```
from Named n, Named m where n.name = m.name
```

Note que as duas últimas consultas requerem mais de um SQL SELECT . Isto significa que a clausula `order by` não ordena corretamente todo o resultado. (Isso também significa que você não pode chamar essas queries usando `Query.scroll()`.)

14.8. A clausula where

A clausula `where` permite estreitar a lista de instancias retornada. Se não houver referencia alguma, pode-se referir a propriedades pelo nome:

```
from Cat where name='Fritz'
```

Se houver uma referência, use o nome da propriedade qualificada:

```
from Cat as cat where cat.name='Fritz'
```

retorna instancias de `Cat` com nome ‘Fritz’.

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

retornará todas as instancias de `Foo`, para cada um que tiver uma instancia de `bar` com a propriedade `date` igual a propriedade `startDate` de `Foo`. Expressões de filtro compostas fazem da clausula `where`, extremamente poderosa. Consideremos:

```
from Cat cat where cat.mate.name is not null
```

Esta query traduzida para uma query SQL `com` uma tabela (inner) join. Se fosse escrever algo como:

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

Poderia-se terminar `com` uma query que necessitasse de join de quatro tabelas, no SQL.

O operador = pode ser uasdo para comparar não apenas propriedades, mas também instancias:

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

A propriedade especial (lowercase) `id` pode ser usada para referenciar o identificador único de um objeto. (Pode-se usar também o nome de sua propriedade)

```
from Cat as cat where cat.id = 123

from Cat as cat where cat.mate.id = 69
```

A segunda query é mais eficiente. Não é necessária nenhuma união de tabelas!

As propriedades de identificadores compostas também podem ser usadas. Suponha que `Person` tenha um identificador composto que consiste de `country` e `medicareNumber`.

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

Mais uma vez, a Segunda query não precisa de nenhum join de tabela.

Assim mesmo, a propriedade especial `class` acessa o valor discriminador da instancia, no caso de persistência polimórfica. O nome de uma classe Java inclusa em uma clausula "where", será traduzida para seu valor descriminante.

```
from Cat cat where cat.class = DomesticCat
```

Pode-se também especificar as propriedades dos components ou tipos de usuário composto (e de componentes de componentes). Nunca tente usar uma expressão de filtro que termine na propriedade de um tipo de componente (ao contrário de uma propriedade de um componente). Por exemplo, se store.owner é uma entidade `com` um componente `address`.

```
store.owner.address.city    // okay
store.owner.address         // error!
```

Um tipo "any" tem as propriedades `id` e `class` especiais, nôs permitindo expressar um join da seguinte forma (onde `AuditLog.item` é uma propriedade mapeada `com <any>`)

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

Veja que `log.item.class` e `payment.class` podem referir-se a valores de colunas de banco de dados completamente diferentes, na query acima.

14.9. Expressões

As expressões permitidas na cláusula `where` inclui a maioria das coisas que você poderia escrever no SQL:

- operadores matemáticos `+, -, *, /`
- operadores de comparação binários `=, >=, <=, <>, !=, like`
- operadores lógicos `and, or, not`
- parenteses `( )`, indicating grouping
- `in`, `not in`, `between`, `is null`, `is not null`, `is empty`, `is not empty`, `member of` and `not member of`

- case "simples" , `case ... when ... then ... else ... end`, and "searched" case, `case when ... then ... else ... end`
- concatenação de string `...||...` ou `concat(...,...)`
- `current_date()`, `current_time()`, `current_timestamp()`
- `second(...)`, `minute(...)`, `hour(...)`, `day(...)`, `month(...)`, `year(...)`,
- qualquer função ou operador definida pela EJB-QL 3.0: `substring()`, `trim()`, `lower()`, `upper()`, `length()`, `locate()`, `abs()`, `sqrt()`, `bit_length()`, `mod()`
- `coalesce()` and `nullif()`
- `str()` para converter valores numericos ou temporais para string
- `cast(... as ...)`, onde o segundo argumento é o nome do tipo hibernate, e`extract(... from ...)` se ANSI `cast()` e `extract()` é suportado pele banco de dados usado
- A função HQL `index()` , que se aplicam a referencias de coleções associadas e indexadas
- As funções hql que retornam expressões de coleções de valores: `size()`, `minelement()`, `maxelement()`, `minindex()`, `maxindex()`, `junto` com o elemento especial, `elements()`, e funções de `índice` que podem ser quantificadas usando `some, all, exists, any, in`.
- Qualquer funçâo escalar pelo bando de dados como `sign()`, `trunc()`, `rtrim()`, `sin()`
- Parametros posicionais ao estilo JDBC `?`
- Parametros nomeados `:name`, `:start_date`, `:x1`
- Literais SQL `'foo'`, `69`, `6.66E+2`, `'1970-01-01 10:00:01.0'`
- Constantes Java `public static final` `ex: Color.TABBY`

`in` e `between` podem ser usadas da seguinte maneira:

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

e as formas negativas podem ser escritas

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

Assim mesmo, `is null` e `is not null` podem ser usados para testar valores nulos.

Booleanos podem ser facilmente usados em expressões, declarando as substituições da HQL query, na configuração do Hibernate

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

Isso irá substituir as palavras chave `true` e `false` `pelos` literais `1` e `0` na tradução do HQL para SQL.

```
from Cat cat where cat.alive = true
```

Pode-se testar o tamanho de uma coleção `com` a propriedade especial `size`, ou a função especial `size()`.

```
from Cat cat where cat.kittens.size > 0
```

```
from Cat cat where size(cat.kittens) > 0
```

Para coleções indexadas, você pode se referir aos índices máximo e mínimo, usando as funções `minindex` e `maxindex`. Similarmente, pode-se referir aos elementos máximo e mínimo de uma coleção de tipos básicos usando as funções `minelement` e `maxelement`.

```
from Calendar cal where maxelement(cal.holidays) > current_date
```

```
from Order order where maxindex(order.items) > 100
```

```
from Order order where minelement(order.items) > 10000
```

As funções SQL `any, some, all, exists, in` são suportadas quando passado o elemento ou o conjunto de índices de uma coleção (`elements` e `indices` de funções), ou o resultado de uma subquery (veja abaixo).

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3 > all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

Note que essas construções - `size`, `elements`, `indices`, `minindex`, `maxindex`, `minelement`, `maxelement`– só podem ser usados na clausula where do Hibernate3.

Elementos de coleções com índice (arrays, lists, maps), podem ser referenciadas pelo índice (apenas na clausula where):

```
from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

A expressão entre colchetes `[]`, pode ser até uma expressão aritimética.

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

O HQL também provê a função interna `index()`, para elementos de associação um-pra-muitos ou coleção de valores.

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

Funções escalares SQL, suportadas pelo banco de dados subjacente.

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

Se ainda ainda não está totalmente convencido, pense o quão maior e menos legível poderia ser a query a seguir, em SQL:

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

Hint: something like

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

14.10. A clausula order by

A lista retornada pela query pode ser ordenada por qualquer propriedade da classe ou componente retornado:

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

As opções `asc` ou `desc` indicam ordem crescente ou decrescente, respectivamente.

14.11. A clausula group by

Uma query que retorne valores agregados, podem ser agrupados por qualquer propriedade de uma classe ou componente retornado:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

Uma clausula `having` também é permitida.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

Funções SQL e funções agregadas são permitidas nas clausulas `having` e `order by`, se suportadas pelo banco de dados subjacente (ex: não no MySQL).

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

Note que, nem a clausula `group by` ou `order by`, podem conter expressões aritiméticas. Veja também que o Hibernate atualmente não expande uma entidade agrupada, então você não pode usar `group por cat` se todas as propriedades de `cat` non-forem agregadas. Você precisa listar todas as propriedades não agregadas explicitamente.

14.12. Subqueries

Para bancos de dados que suportem subselects, o Hibernate suporta subqueries dentro de queries. Uma subquery precisa estar entre parênteses (normalmente uma chamada de função agregada SQL). Mesmo subqueries co-relacionadas (subqueries que fazem referência à alias de outras queries), são aceitas.

```
from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

Note que HQL subqueries podem aparecer apenas dentro de clausulas select ou where.

Para subqueries `com` mais de uma expressão na lista do select, pode-se usar um construtor de tuplas:

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

Veja que em alguns bancos de dados (mas não o Oracle ou HSQL), pode-se usar construtores de tuplas em outros contextos. Por exemplo quando buscando componentes ou tipos de usuário composto.

```
from Person where name = ('Gavin', 'A', 'King')
```

Qual é equivalente ao mais verbalizado:

```
from Person where name.first = 'Gavin' and name.initial = 'A' and name.last = 'King')
```

Há duas razões boas que você pode não querer fazer este tipo de coisa: primeira, não é completamente portável entre plataformas de banco de dados; segunda, a query agora é dependente da ordem de propriedades no documento de mapeamento.

14.13. Exemplos de HQL

As consultas do Hibernate, podem ser muito poderosas e complexas. De fato, o poder da linguagem de consultas é um dos pontos principais na distribuição do Hibernate. Aqui temos algumas consultas de exemplo, muito similares a consultas que usei em um projeto recente. Veja que a maioria das consultas que você irá escrever, são mais simples que estas.

A consultas a seguir retorna o id de order, numero de itens e o valor total do order para todos os orders não pagos para um freguês particular e valor total mínimo dado, ordenando os resultados por valor total. Ao determinar os preços, é usado o catalogo corrente. A query SQL resultante, usando tabelas `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` e `PRICE`, tem quatro inner joins e um (não correlacionado) subselect.

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

Que monstro! Atualmente, na vida real, eu não sou muito afeiçoado a subqueries, então minha query seria mais parecida com isto:

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

A próxima query conta o número de pagamentos em cada status, tirando todos os pagamentos com status `AWAITING_APPROVAL`, onde a mais recente mudança de status foi feita pelo usuário corrente. Traduz-se para uma query SQL `com` dois inner joins e um subselect correlacionado, nas tabelas `PAYMENT`, `PAYMENT_STATUS` e `PAY-`

MENT_STATUS_CHANGE .

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

Se eu tivesse mapeado a Collection `statusChanges` como um List, ao invés de um Set, a query teria sido muito mais simples de escrever.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

A próxima query usa a função `isNull()` do MS SQL Server, para retornar todas as contas e pagamentos não pagos para a organização, para cada usuário corrente pertencente. Traduz-se para uma query SQL `com` três inner joins, um outer join e um subselect nas tabelas `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`,`ACCOUNT_TYPE`, `ORGANIZATION` e `ORG_USER` .

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

Para alguns bancos de dados, precisaremos eleminar o subselect (correlacionado).

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

14.14. update e delete em lote

Agora o HQL suporta declarações, `update`, `delete` e `insert ... select ...` Veja Seção 13.4, “Operações no estilo DML”, para mais detalhes.

14.15. Dicas e Truques

Pode-se contar o número de resultados da query, sem realmente retorna-los.

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue()
```

Para ordenar um resultado pelo tamanho de uma Collection, use a query a seguir.

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

Se seu banco de dados suporta subselects, pode-se colocar uma condição sobre tamanho de seleção na cláusula where da sua query:

```
from User usr where size(usr.messages) >= 1
```

Se seu banco de dados não suporta subselects, use a query a seguir:

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

Com essa solução não se pode retornar um `User com` sem nenhuma menssagem, por causa do "inner join", a forma a seguir também é útil.

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

As propriedades de um JavaBean podem ser limitadas à parâmetros nomeados da query:

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

As Collections são paginaveis, usando a interface `Query com` um filtro:

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

Os elementos da Collection podem ser ordenados ou agrupados usando um filtro de query:

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

Pode-se achar o tamanho de uma Collection sem inicializa-la:

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue();
```

Capítulo 15. Consultas por critérios

O Hibernate provê uma intuitiva e extensível API de critério de query.

15.1. Criando uma instancia `Criteria`

A interface `org.hibernate.Criteria` representa a query ao invés de uma classe persistente particular. A sessão é uma fábrica para intancias de `Criteria` .

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

15.2. Limitando o result set

Um critério individual de query é uma instancia da interface `org.hibernate.criterion.Criterion`. A classe `org.hibernate.criterion.Restrictions` define os métodos da fábrica para obter certos tipos pré fabricados de `Criterion`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

Restrições podem ser logicamente agrupadas.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

Existe um grande número de critérios pré fabricados (subclasses de `Restrictions`), mas um é especialmente útil pois permite especificar o SQL diretamente.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sqlRestriction("lower({alias}.name) like lower(?)", "Fritz%", Hibernate.STRING)
    .list();
```

O parametro `{alias}` será substituido pelo alias da entidade procurada.

Uma maneira alternativa de obter um critério é pegá-lo de uma instancia de `Property`. Você pode criar uma `Property` chamando `Property.forName()`.

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" } ) )
    .list();
```

15.3. Ordenando os resultados

Você pode ordernar os resultados usando `org.hibernate.criterion.Order`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

15.4. Associações

Você pode fácilmente especificar restrições sobre as entidades relacionadas por associações quando estiver navegando usando `createCriteria()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

veja que uma segunda `createCriteria()` retorna um nova instancia de `Criteria`, que faz referencia aos elementos da coleção de `kittens`.

A segunte forma alternativa e mais indicada em certas circunstancias

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` não irá criar uma nova instancia de `Criteria`.)

Veja que as coleções de kittens contidas pelas instancias de `Cat` retornadas pelas duas consultas anteriores *não* estão pre-filtrados pelos critérios! Se você desejar recuperar apenas os kittens que combinam com os critérios, você deve usar um `ResultTransformer`.

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

15.5. Recuperação dinamica de associações

Você pode especifica a semantica de recuperação de associações em tempo de execução usando `setFetchMode()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

Esta consulta retornará `mate` e `kittens` através de um outer join. Veja Seção 19.1, “Estratégias de Fetching” para mais informações.

15.6. Consultas por exemplo

A classe `org.hibernate.criterion.Example` permite a criação de uma consulta de critérios a partir de um instancia passada

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

As propriedades de versão, os identificadores e as associações são ignorados. Por default, as propriedades nulas são excluídas.

Você pode ajustar como o `Example` é aplicado.

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
    .ignoreCase()              //perform case insensitive string comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

Você pode inclusive usar exemplos par aplicar critérios em objetos associados.

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

15.7. Projections, aggregation and grouping

A classe `org.hibernate.criterionProjections` é uma fábrica para instancias de `Projection`. Nós aplicamos uma projeção a uma consulta chamando `setProjection()`.

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

Não é necessário nenhum "group by" explicito em uma consulta por critérios. Certo tipos da projeções são definidos como *projeções agrupadas*, que também na aparecem cláusula `group by` do SQL.

Opcionalmente um alias pode ser atribuído a uma projeção, de modo que o valor projetado possa ser referenciado em restrições ou e ordenações. Eis as duas maneiras diferentes de fazer isto:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"), "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr") )
    .addOrder( Order.asc("colr") )
    .list();
```

Os métodos `alias()` e `as()` simplesmente envolvem uma instancia de projection em outra instancia de `Projection`. Como um atalho, você pode atribuir um alias quando você adiciona a projeção a uma lista da projeção:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

Você pode também usar `Property.forName()` expressar projeções:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

15.8. Consultas e sub consultas separadas

A classe `DetachedCriteria` permite que uma consulta seja criada fora do escopo de uma sessão, e então executá-la mais tarde usando uma `Session` qualquer.

```
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResults(100).list();
txn.commit();
session.close();
```

Ums `DetachedCriteria` também pode ser usada para expressar uma subquery. As instancias de critérios que envolvem subqueries podem ser obtidos através de `Subqueries` ou de `Property`.

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight).gt(avgWeight) )
    .list();
```

```
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

Até mesmo subqueries correlacionados são possíveis:

```
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.class, "cat2")
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight).gt(avgWeightForSex) )
    .list();
```

15.9. Consultas por identificador natural

Para a maioria das consultas, incluindo consultas por critérios, o cache de consultas não é muito eficiente, porque a invalidação do chache de consultas ocorre com muita frequencia. Entretanto, há um tipo especial da con-

sultas onde nós podemos optimizar o algoritmo do invalidation de cache: lookups para chave naturais constantes . Em algumas aplicações, este tipo da consulta ocorre com frequencia. A API critérios fornece métodos especiais para estes casos.

Primeiro, você deve mapear a chave natural da sua da entidade usando `< natural-id>` , e habilitar o uso do cache de segundo-nível.

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
</class>
```

Veja que esta funcionalidade não foi desenvolvida para o uso com entidades com chaves naturais *mutáveis*.

Depois, habilite o cache de consultas do Hibernate.

Agora, `Restrictions.naturalId()` permite que empreguemos um algoritmo mais eficiente para o cache.

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

Capítulo 16. SQL nativo

Você também pode criar queries no dialeto SQL nativo de seu banco de dados. Isto é útil se você quiser utilizar características específicas do banco de dados tais como hints ou a palavra chave `CONNECT` do Oracle. Também pode ser um caminho de migração facil de uma aplicação baseada em SQL/JDBC para uma aplicação Hibernate.

O Hibernate3 permite especificar SQL escrito manualmente(incluindo estored procedure) para todas as operações criação, atualização, deleteção e recuperação.

16.1. Usando o `SQLQuery`

A execução de consultas SQL nativas é controlada pela interface `SQLQuery`, que é obtida chamando `Session.createSQLQuery()`. Abaixo descrevemos como usar esta API para consultas.

16.1.1. Consultas escalres

A consulta SQL mais básica é adquirir uma lista escalar (valores).

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

Ambos retornarão um List de arrays de Objetc (Objet[]) com valores escalares por cada coluna da tabela CATS. O Hibernate usará ResultSetMetadata para deduzir a ordem atual e os tipos dos valores escalares retornados.

Para evitar o overhead ao se usar `ResultSetMetadata` ou simplesmente para ser mais explícito no que é retornado você podem usar `addScalar()`.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

Esta query especifica:

- A String que contém a query SQL
- as colunas e tipos de retorno

Isso ainda retornará um array de Objetcs, mas não será usado `ResultSetMetdata` e sim, tentará tratar explicatamente as colunas ID, NAME e BIRTHDATE respectivamente como Long, String e Short no resultset retornado. Também significa que somente estas três colunas serão devolvidas, embora a consulta esteja usando `*` e poderia devolver mais que as três colunas listadas.

É possível omitir a informação de tipo para todas ou apenas algumas colunas.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

Esta é essencialmente a mesma consulta deantes, mas agora `ResultSetMetaData` é usado para decidir o tipo de NAME e BIRTHDATE,e o tipo de ID é especificado explicitamente.

Como os java.sql.Types retornados por `ResultSetMetdata` são mapeados para os tipos do Hibernate é controlado pelo Dialeto. Se um tipo específico não estiver mepeando ou não resulta no tipo esperado é possível personalizar isto via chamadas a `registerHibernateType` no Dialeto.

16.1.2. Entity queries

As consultas anteriores são usadas para retornar valores escalares, basicamente devolvendo os valores "crus" do resultset. Os exemplos seguintes mostram como recuperar objetos de entidade de um sql nativo através de `addEntity()`.

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

Esta query especifica:

A String que contém a query SQL

- A entidade retornada pela query

Assumindo que Cat é mapeado como uma classe com as colunas ID, NOME e BIRTHDATE as anteriores consultas devolverão uma Lista onde cada elemento é uma entidade de Cat.

Se a entidade for mapeada como `many-to-one` para outra entidade é exigido que a consulta devolva também esta outra entidade, caso contrário acontecerá um erro de banco de dados "coluna não encontrada". As colunas adicionais serão devolvidas automaticamente ao se usar a anotação notação `*`, mas nós preferimos explícitar como no exemplo seguinte `many-to-one` para um `Dog`:

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

Isso fará com que cat.getDog() passe a afuncionar corretamente.

16.1.3. Manipulando associações e coleções

É possível associar antecipadamente o `Dog` para evitar uma possível ida ao banco para inicializar o proxy. Isto é feito através do método de `addJoin()` que lhe permite fazer associações em uma associação ou coleção.

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d WHERE c.DO
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

Neste exemplo o `Cat`'s devolvido terá sua propriedade `dog` completamente inicializada sem qualquer roundtrip extra ao banco de dados. Veja que nós demos um alias ("cat") para poder usar a propriedade no join. É possível fazer a mesma associação para coleções, por exemplo se `Cat` tivesse um associação one-to-many pata `Dog`.

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE c.ID =
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

Neste pontonós estamos alcançando os limites do que é possível fazer com consultas nativas sem começar a aumentar o sql para que o mesmo seja utilizável dentro Hibernate; os problemas começam a surgir ao setretornar entidades múltiplas do mesmo tipo ou quando o alias para as colonas não é o bastante.

16.1.4. Retornando múltiplas Entidades

Até agora assumimos que os nomes das colunas no result set são os mesmo do documento de mapeamento. Isto pode ser um problemapara consultas SQL que façam join entra múltiplas tabelas, dado que os mesmos nomes de coluna podem aparecer em mais de uma tabela.

Nesta consulta(que provavelmente falhará), é necessário a injeção de alias para colunas:

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

A intenção para esta consulta é devolver duas instancias de Cat por linha, um gato e sua mãe. Isto falhará pois há um conflito de nomes pois eles estão mapeados com os mesmos nomes de coluna e em alguns bancos de dados os aliases de coluna devolvidos provável estarão na forma "c.ID", "C.NAME", etc. que não são iguais aos especificedos

A forma seguinte não é vulnerável a duplicação de nomes de coluna:

```
sess.createSQLQuery("SELECT {cat.*}, {mother.*}  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

Esta consulta especificou:

- A string que contém a SQL query com os locais para o Hibernate injetar os pseudônimos das colunas
- as entidades devolvidas pela consulta

A anotação {cat.*} e {mother.*} usada acima é um atalho pata "todas as propriedades." Alternativamente, você pode listar explicitamente as colunas, mas até mesmo neste caso nós deixamos o Hibernate injetar os aliases SQL das colunas para cada propriedade. O local para um alias de coluna é o nome de propriedade qualificado pelo alias da tabela. No exemplo seguinte, nós recuperamos Cats e suas mothers de uma tabela diferente (cat_log) do que o declarado no mapeaemento. Veja que nós podemos usar os aliases de propriedade até mesmo na cláusula where se quisermos.

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
         "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mother.*} " +
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

16.1.4.1. Alias and property references Aliases e referências de propriedade

Para a maioria dos casos acima é necessária injeção de aliases, mas para consultas relativo a mapeamentos mais complexos como propriedades compostas, discriminadores de herança, coleções etc. há alguns aliases específicos para se usar para permitir que o Hibernate injete os próprios aliases.

A mesa seguinte tabela mostra as diferentes possibilidades de usar a injeção de alias. Nota: os nomes dos alias no resultado são exemplos, cada alias terá um nome unico e provavelmente diferente quando usado.

Tabela 16.1. Alias injection names

Descrição	Sintaxe	Exemplo
Uma propriedade simples	`{[aliasname].[propertyname]`	`A_NAME as {item.name}`
Uma propriedade composta	`{[aliasname].[componentname].[propertyname]}`	`CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}`
O Discritor de uma entidade	`{[aliasname].class}`	`DISC as {item.class}`
Todas as propriedades de uma entidade	`{[aliasname].*}`	`{item.*}`
Uma chave de coleção	`{[aliasname].key}`	`ORGID as {coll.key}`
O id de uma coleção	`{[aliasname].id}`	`EMPID as {coll.id}`
O elemento de uma coleção	`{[aliasname].element}`	`XID as {coll.element}`
propriedade de um elemento na coleção	`{[aliasname].element.[propertyname]}`	`NAME as {coll.element.name}`
Todas as propriedades do elemento na coleção	`{[aliasname].element.*}`	`{coll.element.*}`
Todas as propriedades do a coleção	`{[aliasname].*}`	`{coll.*}`

16.1.5. Retornando Entidades não gerenciads

É possível aplicar um ResultTransformer a consultas SQL nativas. Permitindo por exemplo o retorno de entidades não gerenciadas.

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

Esta query especifica:

- A string que contém a query SQL
- Um transformer para o resultado

A constulta anterior retornará uma lista de `CatDTO` que foi instanciada e injetada com os valores de NOME e BIRTHNAME em suas propriedades correspondentes ou campos.

16.1.6. Controlando a herança

Consultas sql nativas que retornam entidades que são mapeadas como parte de uma herança tem que incluir todas as propriedades para a clase base e todas as propriedades da subclase.

16.1.7. Parâmetros

Consultas sql nativas suportam parâmetros positionais e também parâmetros com nomes:

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like ?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like :name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

16.2. Consultas SQL com nomes

Podem ser definidas consultas SQL com nomes no documento de mapeamento e elas podem ser chamadas exatamente do mesmo modo como uma consulta de HQL com nome. Neste caso, nós não precisamos chamar*not*.

```
<sql-query name="persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

Os elementos `<return-join>` e `<load-collection>` são usados para fazer associações e definir as consultas que inicializam coleções, respectivamente.

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

Uma consulta SQL com nome pode devolver um valor de escalar. Você precisa declarar o alias da coluna e o tipo do Hibernate usando o elemento `<return-scalar>`:

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

Você pode externalizar as informações de mapeamento do resultset em um elemento `<resultset>` para poder

usar-lo em várias consultas SQL com nome ou com a API `setResultSetMapping()`

```
<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

Você pode ainda usar as informações do mamepamento de resultset mapeadas nos arquivos hbm diretamente no código de java.

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

16.2.1. Usando a propriedade retornada para especificar explicitamente os nomes de colunas e aliás

Com `<return-property>` você pode dizer explicitamente ao Hibernate que aliases da coluna ele deve usar, em vez de usar a sintaxe `{}`-syntax para deixar Hibernate injetar seus próprios aliases.

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
        <return-property name="name" column="myName"/>
        <return-property name="age" column="myAge"/>
        <return-property name="sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

`<return-property>` também trabalhos com colunas múltiplas. Isto resolve alimitação da sintaxe `{}` que não permite um controle fino de propriedades multi colunas.

```
<sql-query name="organizationCurrentEmployments">
    <return alias="emp" class="Employment">
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
        <return-property name="endDate" column="myEndDate"/>
    </return>
        SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.employer},
        STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
        REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE, CURRENCY
        FROM EMPLOYMENT
        WHERE EMPLOYER = :id AND ENDDATE IS NULL
```

```
        ORDER BY STARTDATE ASC
</sql-query>
```

Veja que neste exemplo nós usamos `<return-property>` combinado com a a sintaxe `{}` para a injeção. Permitindo que os usuários escolham como querem se refrenciar a coluna e as propriedades.

Se seu mapeamento tiver um discriminador você deve usar `<return-discriminator>` para especificar a coluna do discriminador.

16.2.2. Usando stored procedures para consultas

Hibernate 3 introduz o suporte a consultas por stored procedures e funções. A maioria da documentação seguinte é equivalente para ambos. As stored procedures / função tem que devolver um resultset como o primeiro parâmetro de retorno para serem usadaos pelo Hibernate. Um exemplo disso e são as stored function no Oracle 9 e superior:

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT EMPLOYEE, EMPLOYER,
 STARTDATE, ENDDATE,
 REGIONCODE, EID, VALUE, CURRENCY
 FROM EMPLOYMENT;
      RETURN  st_cursor;
 END;
```

Para usar esta consulta dentro do Hibernate você precisa mapear através de uma consulta com nome.

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

Veja que atualmente as stored procedures apenas podem retornar valores escalares e entidades. `<return-join>` e `<load-collection>` não são suportados.

16.2.2.1. Regras/limitações no uso de stored procedures

Para usar stored procedures no Hibernate os stored procedures/functions têm que seguir algumas regras. Se não seguirem essas regras não poderã ser usadas com o Hibernate. Se mesmo assim você quiser usar essa stored procedures você tem que executá-los via `session.connection()`. As regras são diferentes para cada banco de dados, visto que cada fornecedor de banco de dados têm semântica/sintaxe diferentes para stored procedures.

Stored procedures não podem ser paginadas com `setFirstResult()/setMaxResults()`.

É recomendado a forma de chmanda padrão do SQL92: `{ ? = call functionName(<parameters>) }` ou `{ ? = call procedureName(<parameters>}`. Sintaxe nativas de chamadas não são suportadas.

Para o Oracle se aplicam as seguintes regras:

- A funçã deve retornar um result set. O primeiro parâmetro da stored procedure dever ser uma `OUT` que retorne um result set. Isto é feito usando o tipo `SYS_REFCURSOR` no Oracle 9 ou 10. No Oracle é necessário definir o tipo `REF CURSOR`, veja a documentação do Oracle.

Para o Sybase ou MS SQL server se aplicam as seguintes regras:

- A procedure deve retornar um result set. Veja que estes servidores pode retornar múltiplos result sets e update counts. O Hibernate ira iterar os resultados e pegar o primeiro resultado que é o valor de retorno do result set. O resto será descartado.
- Se você habilitar `SET NOCOUNT ON` na sua procedure, ela provavelmente será mais eficiente. Mas, isto não é obrigatório

16.3. SQL customizado para create, update e delete

Hibernate3 pode usar SQL customizado para operações de create, update e delete. A persistência de classe e collection no Hibernate já contem alguma strings de configurações (insertsql, deletesql, updatesql etc.). O mapaemento das tags `<sql-insert>`, `<sql-delete>`, e `<sql-update>` sobreecreve essas strings:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ? )</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

O SQL é executado diretamente no seu banco de dados, então você pode usar qualquer linguagem que quiser. Isto com certeza reduzira a portabilidade do seu mapeamento se você utilizar um SQL para um banco de dados especifico.

Stored Procedures são suportadas se o atributo the `callable` estiver ativado:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert callable="true">{call createPerson (?, ?)}</sql-insert>
    <sql-delete callable="true">{? = call deletePerson (?)}</sql-delete>
    <sql-update callable="true">{? = call updatePerson (?, ?)}</sql-update>
</class>
```

A ordem de posições dos parâmetros são vitais, pois eles devem estar na mesma seqüência esperada pelo Hibernate.

Você pode ver a ordem esperada ativando o debug logging no nível `org.hibernate.persister.entity`. Com este nível ativado, o Hibernate irá imprimir o SQL estático que foi usado para create, update, delete, etc. Entidades. (Para ver a seqüência esperada, lembre-se de não incluir seu SQL customizado no arquivo de mapeamento, pois ele irá sobreecreve o SQL estático gerado pelo Hibernate).

As stored procedures são na maioria dos casos (leia-se: melhor não fazer) requeridas para retornar o numero de linhas inseridas/atualizadas/deletadas. O hibernate tem algumas verificações em tempo de execução para o sucesso da declaração. O Hibernate sempre registra o primeiro parâmetro da declaração como uma saída numérica para operações CRUD.

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VARCHAR2)
    RETURN NUMBER IS
BEGIN

    update PERSON
    set
        NAME = uname,
    where
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

16.4. SQL customizado para carga

Você pode declarar sua própria query SQL (ou HQL) para iniciar entidades:

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

Este é apenas uma declaração de query com nome, como discutido anteriormente. Você pode referenciar esta query com nome em um mapeamento de classe:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

Isto também funciona com stored procedures.

Você pode tembém definir uma query para iniciar uma collection:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

Você pode definir um loader de entidade que carrege uma coleção através de join fetching:

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

Capítulo 17. Filtrando dados

O Hibernate3 provê um novo método inovador para manusear dados com regras de "visibilidade". Um *Filtro do Hibernate* é um filtro global, nomeado e parametrizado que pode se habilitado ou não dentro de um Session do Hibernate.

17.1. Filtros do Hibernate

O Hibernate tem a habilidade de pré definir os critérios do filtro e anexar esses filtros no nível da classe e no nível da coleção. Um critério do filtro é a habilidade de definir uma cláusula restritiva muito semelhante ao atributo "where" disponível para a classe e várias coleções. A não ser que essas condições de filtros podem ser parametrizadas. A aplicação pode, então, fazer uma decisão em tempo de execução se os filtros definidos devem estar habilitados e quais valores seus parâmetros devem ter. Os filtros podem ser usados como Views de bancos de dados, mas com parametros internos à aplicação.

Para usar esses filtros, eles primeiramente devem ser definidos e anexados aos elementos do mapeamento apropriados. Para definir um filtro, use o elemento `<filter-def/>` dentro do elemento `<hibernate-mapping/>`:

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

Então esse filtro pode ser anexo à uma classe:

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</class>
```

ou em uma coleção:

```
<set ...>
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</set>
```

ou mesmo para ambos (ou muitos de cada) ao mesmo tempo.

Os métodos na `Session` são: `enableFilter(String filterName)`, `getEnabledFilter(String filterName)`, e `disableFilter(String filterName)`. Por padrão, os filtros não são habilitados dentro de qualquer session; Eles devem ser explicitamente habilitados usando o método `Session.enabledFilter()`, que retorna uma instância da interface `Filter`. Usando o filtro simples definido acima, o código se pareceria com o seguinte:

```
session.enableFilter("myFilter").setParameter("myFilterParam", "some-value");
```

Veja que os métodos da interface org.hibernate.Filter permite o encadeamento de funções, comum à maioria das funções do Hibernate.

Um exemplo completo, usando dados temporais com um padrão efetivo de registro de datas:

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
...
```

```
    <many-to-one name="department" column="dept_id" class="Department"/>
    <property name="effectiveStartDate" type="date" column="eff_start_dt"/>
    <property name="effectiveEndDate" type="date" column="eff_end_dt"/>
...
    <!--
        Note that this assumes non-terminal records have an eff_end_dt set to
        a max db date for simplicity-sake
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
...
    <set name="employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
    </set>
</class>
```

Para garantir que você sempre tenha registro efetivos, simplesmente habilite o filtro na session antes de recuperar os dados dos empregados:

```
Session session = ...;
session.enabledFilter("effectiveDate").setParameter("asOfDate", new Date());
List results = session.createQuery("from Employee as e where e.salary > :targetSalary")
         .setLong("targetSalary", new Long(1000000))
         .list();
```

No HQL acima, mesmo que mencionamos apenas uma restrição de salário nos resultados, por causa do filtro habilitado, a consulta retornará apenas os funcionários ativos cujo salário é maior que um milhão de dólares.

Nota: se você planeja usar filtros com outer join (por HQL ou por load fetching) seja cuidadoso na direção da expressão de condição. É mais seguro configura-lo com para um left outer join; geralmente, coloque o parâmetro primeiro seguido pelo nome da coluna após o operador.

Após um filtros ser definido, ele pode ser associado a múltiplas entidades e/ou coleções cada uma com sua própria condição. Isso pode ser tedioso quando as circunstâncias são as mesmas para todos os casos. Assim `<filter-def/>` permite definir uma condição padrão, como um atributo ou um CDATA:

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-def>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

Esta condição default será usada sempre que o filtro for associado a um elemento sem especificar uma condição. Veja que isto significa que você pode especificar uma condição específica que substitui a condição default para esse caso em particular como parte do processo de associação do filtro.

Capítulo 18. Mapeamento XML

Veja que essa é uma feature experimental no Hibernate 3.0 e o desenvolvimento esta bastante ativo.

18.1. Trabalhando com dados em XML

O Hibernate permite que se trabalhe com dados persistentes em XML quase da mesma maneira como você trabalhar com POJOs persistentes. Uma árvore XML parseada, pode ser imaginada como apenas uma maneira de representar os dados relacionais como objetos, ao invés dos POJOs.

O Hibernate suporta a API dom4j para manipular árvores XML. Você pode escrever queries que retornem árvores dom4j do banco de dados e automaticamente sincronizar com o banco de dados qualquer modificação feita nessas árvores. Você pode até mesmo pegar um documento XML, parsear usando o dom4j, e escrever as alterações no banco de dados usando quaisquer operações básicas do Hibernate: `persist()`, `saveOrUpdate()`,`merge()`, `delete()`, `replicate()` (merging ainda não é suportado)

Essa funcionalidade tem várias aplicações incluindo importação/exportação de dados, externalização de dados de entidade via JMS or SOAP e relatórios usando XSLT.

Um mapeamento simples pode ser usado para simultaneamente mapear propriedades da classe e nós de um documento XML para um banco de dados ou, se não houver classe para mapear, pode ser usado simplesmente para mapear o XML.

18.1.1. Especificando o mapeamento de uma classe e de um arquivo XML simultaneamente

Segue um exemplo de como mapear um POJO e um XML ao mesmo tempo:

```
<class name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="accountId"
            column="ACCOUNT_ID"
            node="@id"/>

    <many-to-one name="customer"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"/>

    <property name="balance"
            column="BALANCE"
            node="balance"/>

    ...

</class>
```

18.1.2. Especificando somente um mapeamento XML

Segue um exemplo que não contém uma classe POJO:

```
<class entity-name="Account"
        table="ACCOUNTS"
        node="account">
```

```
    <id name="id"
            column="ACCOUNT_ID"
            node="@id"
            type="string"/>

    <many-to-one name="customerId"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"
            entity-name="Customer"/>

    <property name="balance"
            column="BALANCE"
            node="balance"
            type="big_decimal"/>

    ...

</class>
```

Esse mapeamento permite que você acesse os dados como uma árvore dom4j ou um grafo de de pares nome de propriedade/valor (`Maps` do Java). Os nomes de propriedades são somente construções lógicas que podem ser referenciadas em consultas HQL.

18.2. Mapeando metadados com XML

Muitos elementos do mapeamento do Hibernate aceitam o atributo `node`. Por meio dele, você pode especificar o nome de um atributo ou elemento XML que contém a propriedade ou os dados da entidade. O formato do atributo `node` deve ser o seguinte:

- `"element-name"` - mapeia para o elemento XML com determinado nome
- `"@attribute-name"` - mapeia para o atributo XML com determinado nome
- `"."` - mapeia para o elemento pai
- `"element-name/@attribute-name"` - mapeia para para o atributo com determinado nome do elemento com determinado nome

Para coleções e associações simples, existe o atributo adicional `embed-xml`. Se o atributo `embed-xml="true"`, que é o valor padrão, a árvore XML para a entidade associada (ou coleção de determinado tipo de valor) será embutida diretamente na árvore XML que contém a associação. Por outro lado, se `embed-xml="false"`, então apenas o valor do identificador referenciado irá aparecer no XML para associações simples e coleções simplesmentenão irão aparecer.

Você precisa tomar cuidado em não deixar o`embed-xml="true"` para muitas associações, pois o XML não suporta bem referências circulares.

```
<class name="Customer"
        table="CUSTOMER"
        node="customer">

    <id name="id"
            column="CUST_ID"
            node="@id"/>

    <map name="accounts"
            node="."
            embed-xml="true">
        <key column="CUSTOMER_ID"
                not-null="true"/>
        <map-key column="SHORT_DESC"
                node="@short-desc"
```

```
                type="string"/>
        <one-to-many entity-name="Account"
                embed-xml="false"
                node="account"/>
    </map>

    <component name="name"
            node="name">
        <property name="firstName"
                node="first-name"/>
        <property name="initial"
                node="initial"/>
        <property name="lastName"
                node="last-name"/>
    </component>

    ...

</class>
```

Nesse caso, decidimos embutir a colenção de account ids, e não os dados de accounts. A query HQL a seguir:

```
from Customer c left join fetch c.accounts where c.lastName like :lastName
```

Retornaria um conjunto de dados como esse:

```
<customer id="123456789">
    <account short-desc="Savings">987632567</account>
    <account short-desc="Credit Card">985612323</account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

Se você setar `embed-xml="true"` em um mapeamento `<one-to-many>`, os dados se pareceriam com o seguinte:

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings">
        <customer id="123456789"/>
        <balance>100.29</balance>
    </account>
    <account id="985612323" short-desc="Credit Card">
        <customer id="123456789"/>
        <balance>-2370.34</balance>
    </account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

18.3. Manipulando dados em XML

Vamos reler e atualizar documentos em XML em nossa aplicação. Nós fazemos isso obtendo uma session do dom4j:

```
Document doc = ....;

```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts where c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}

tx.commit();
session.close();
```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId);
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

tx.commit();
session.close();
```

É extremamente útil combinar essa funcionalidade com a operação `replicate()` do Hibernate para implementar importação/exportação baseadas em XML.

Capítulo 19. Aumentando a performance

19.1. Estratégias de Fetching

Uma *estratégia de fetching* é a estratégia que o Hibernate irá usar para buscar objetos associados se a aplicação precisar navegar pela associação. Estratégias de Fetch podem ser declaradas nos metadados de mapeamento O/R, ou sobrescritos por uma query HQL ou query com `Criteria`.

O Hibernate3 define as seguintes estratégias de fetching:

- *Join fetching* - o Hibernate busca o objeto ou coleção associada no mesmo `SELECT`, usando um `OUTER JOIN`.
- *Select fetching* - um segundo `SELECT` é usado para buscar a entidade ou coleção associada. A menos que você desabilite lazy fetching especificando `lazy="false"`, esse segundo SELECT será executado apenas quando você acessar a associação.
- *Subselect fetching* - um segundo `SELECT` será usado para buscar as coleções associadas de todas as entidades buscadas na query ou fetch anterior. A menos que você desabilite lazy fetching especificando `lazy="false"`, esse segundo SELECT será executado apenas quando você acessar a associação.
- *Batch fetching* - uma opção de otimização para o Select Fetching – O Hibernate busca um lote de instâncias ou entidades usando um único `SELECT`, especificando uma lista de primary keys ou foreing keys.

O Hibernate distingue também entre:

- *Immediate fetching* - uma associação, coleção ou atributo é buscado como ela é carregada (Qual SQL é usado). Não se confuda com eles! Nós usamos fetch para melhorar a performance. Nós podemos usar lazy para definir um contrato para qual dado é sempre disponível em qualquer instância desanexada de uma classe qualquer. imediatamente, quando o pai é carregado.
- *Lazy collection fetching* - a coleção é buscada quando a aplicação invoca uma operação sobre aquela coleção (Esse é o padrão para coleções)
- *"Extra-lazy" collection fetching* - elementos individuais de uma coleção são acessados do banco de dados quando preciso. O Hibernate tenta não buscar a coleção inteira dentro da memória ao menos que seja absolutamente preciso. (indicado para coleções muito grandes)
- *Proxy fetching* - uma associação de um valor é carregada quando um método diferente do getter do identificador é invocado sobre o objeto associado.
- *"No-proxy" fetching* - uma associação de um valor é carregada quando a variável da instância é carregada. Comparada com a proxy fetching, esse método é menos preguiçoso (lazy)(a associação é carregada somente quando o identificador é acessada) mas é mais transparente, já que não há proxies visíveis para a aplicação. Esse método requer instrumentação de bytecodes em build-time e é raramente necessário.
- *Lazy attribute fetching* - um atributo ou associação de um valor é carregada quanto a varíavel da instância é acessada. Esse método requer instrumentação de bytecodes em build-time e é raramente necessário.

Nós temos aqui duas noções ortogonais: *quando* a associação é carregada e *como* ela é carregada (Qual SQL é usado). Não se confuda com eles! Nós usamos `fetch` para melhorar a performance. Nós podemos usar lazy para definir um contrato para qual dado é sempre disponível em qualquer instância desconectada de uma classe

qualquer.

19.1.1. Trabalhando com associações preguiçosas (lazy)

Por padrão, o Hibernate3 usa busca preguiçosa para coleções e busca preguiçosa com proxy para associações de um valor. Esses padrões fazem sentido para quase todas as associações em quase todas a aplicações.

Veja: se você setar `hibernate.default_batch_fetch_size`, O Hibernate irá usar otimização de carregamento em lote para o carregamento preguiçoso(Essa otimização pode ser também habilitada em um nível mais fino).

Porém, a busca preguiçosa tem um problema que você precisar saber. Acesso a associações preguiçosas fora do contexto de uma sessão aberta do Hibernate irá resultar numa exceção. Por exemplo:

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Error!
```

Como a coleção de permissões não foi inicializada quando a `Session` foi fechada, a coleção não poderá carregar o seu estado. O Hibernate não suporta inicialização tardia ara objetos desconectados. Para resolver isso, é necessário mover o código que carrega a coleção para antes da transação ser comitada.

Alternativamente, nós podemos usar uma coleção ou associação com carga antecipada, especificando `lazy="false"` no mapeamento da associação. Porém, a intenção é que a inicialização tardia seja usada por quase todas as coleções e associações. Se você definir muitas associações com carga antecipada em seu modelo de objetos, o Hibernate irá precisar carregar o banco de dados inteiro na memória em cada transação!

Por outro lado, nós geralmente escolhemos join fetching (que é não preguiçosa por natureza) ao invés de select fetching em uma transação particular. Nós iremos ver como customizar a estratégia de busca. No Hibernate3, os mecanismos para escolher a estratégia de fetching são identicos para as associações simples e para coleções.

19.1.2. Personalizando as estratégias de recuperação

O select fetching (o padrão) é extremamente vunerável para problemas em select N+1 , então nós iremos querer habilitar o join fetching no documento de mapeamento:

```
<set name="permissions"
           fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

A estratégia de `fetch` definida no documento de mapeamento afeta:

- recupera via `get()` ou `load()`
- Recuperações que acontecem implicitamente quando navegamos por uma associação

- `Criteria` queries
- buscas por HQL se buscar por `subselect` for usado

Independentemente da estratégia de busca que você usar, o grafo não preguiçoso definido será garantidamente carregado na memória. Note que isso irá resultar em diversos selects imediatos sendo usados em um HQL em particular.

Usualmente não usamos documentos de mapeamento para customizar as buscas. Ao invés disso, nós deixamos o comportamento padrão e sobrescrevemos isso em uma transação em particular, usando `left join fetch` no HQL. Isso diz ao Hibernate para buscar a associação inteira no primeiro select, usando um outer join. Na API de busca `Criteria`, você irá usar `setFetchMode(FetchMode.JOIN)`.

Se você quiser mudar a estratégia de busca usada pelo `get()` ou `load()`, simplesmente use uma query `Criteria`, por exemplo:

```
User user = (User) session.createCriteria(User.class)
                .setFetchMode("permissions", FetchMode.JOIN)
                .add( Restrictions.idEq(userId) )
                .uniqueResult();
```

(Isto é o equivalente do Hibernate para o que algumas soluções ORM chamam de "plano de busca")

Um meio totalmente diferente de evitar problemas com selects N+1 é usar um cache de segundo nível.

19.1.3. Proxies de associação single-ended

A recuperação tardia para coleções é implementada usando uma implementação própria do Hibernate para coleções persistentes. Porém, um mecanismo diferente é necessário para comportamento tardio para associações de um lado só. A entidade alvo da associação precisa usar um proxy. O Hibernate implementa proxies para inicialização tardia em objetos persistentes usando manipulação de bytecode (via a excelente biblioteca CGLIB).

Por padrão, o Hibernate3 gera proxies (na inicialização) para todas as classes persistentes que usem eles para habilitar recuperaçãopreguiçosa de associações `many-to-one` e `one-to-one`.

O arquivo de mapeamento deve declaram uma interface para usar como interface de proxy para aquela classe, com o atributo `proxy`. Por padrão, o Hibernate usa uma subclasse dessa classe. *Note que a classe a ser usada via proxy precisa implementar o construtor padrão com pelo menos visibilidade de package. Nós recomendamos esse construtor para todas as classes persistentes!*

Existe alguns truques que você deve saber quando extender esse comportamento para classes polimórficas, dessa maneira:

```
<class name="Cat" proxy="Cat">
    ......
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>
```

Primeiramente, instâncias de `Cat` nunca serão convertidas para `DomesticCat`, mesmo que a instância em questão seja uma estância de `DomesticCat`:

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a proxy (does not hit the db)
if ( cat.isDomesticCat() ) {                  // hit the db to initialize the proxy
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
```

```
}
```

É possível quebrar o proxy ==.

```
Cat cat = (Cat) session.load(Cat.class, id);           // instantiate a Cat proxy
DomesticCat dc =
        (DomesticCat) session.load(DomesticCat.class, id);  // acquire new DomesticCat proxy!
System.out.println(cat==dc);                           // false
```

Porém a situação não é tão ruim como parece. Mesmo quando temos duas referências para objetos proxies diferentes, a instância deles será o mesmo objeto

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

Terceiro, Você não pode usar um proxy CGLIB em uma classe `final` ou com qualquer método `final`.

Finalmente, se o seu objeto persistente adquirir qualquer recursto durante a instanciação (em inicializadores ou construtor padrão), então esses recursos serão adquiridos pelo proxy também. A classe de proxy é uma subclasse da classe persistente.

Esses problemas são todos devido a limitação fundamental do modelo de herança simples do Java. Se você quiser evitar esse problemas em suas classes persistentes você deve imeplementar uma interface que declare seus métodos de negócio. Você deve especificar essas interfaces no arquivo de mapeamento. Ex:

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

onde `CatImpl` implementa a interface `Cat` e `DomesticCatImpl` implementa a interface `DomesticCat`. Então proxies para instâncias de `Cat` e `DomesticCat` serão retornadas por `load()` ou `iterate()`. (Note que `list()` geralmente não retorna proxies).

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.iterate("from CatImpl as cat where cat.name='fritz'");
Cat fritz = (Cat) iter.next();
```

Relacionamentos são também carregados preguiçosamente. Isso significa que você precisa declarar qualquer propriedade como sendo do tipo `Cat`, e não `CatImpl`.

Algumas operações *não* requerem inicialização por proxy:

- `equals()`, se a classe persistente não sobrescrever `equals()`
- `hashCode()`, se a classe persistente não sobrescrever `hashCode()`
- O método getter do identificador

O Hibernate irá detectar classes persistentes que sobrescrevem `equals()` ou `hashCode()`.

Escolhendo `lazy="no-proxy"` ao invés do padrão `lazy="proxy"`, podemos evitar problemas associados com typecasting. Porém, iremos precisar de instrumentação de bytecode em tempo de compilação e todas as operações irão resultar em iniciazações de proxy imediatas.

19.1.4. Inicializando coleções e proxies

Será lançada uma `LazyInitializationException` se uma coleção não inicializada ou proxy é acessado fora do escopo da `Session`, isto é, quando a entidade que contém a coleção ou tem a referência ao proxy estiver no estado destachado.

Algumas vezes precisamos garantir qie o proxy ou coleção é inicializado antes de se fechar a `Session`. Claro que sempre podemos forçar a inicialização chamando `cat.getSex()` ou `cat.getKittens().size()`, por exemplo. Mas isto parece confuso para quem lê o código e não é conveniente para códigos genéricos.

Os métodos estáticos `Hibernate.initialize()` e `Hibernate.isInitialized()` possibilitam a aplicação uma maneira conveniente de trabalhar com coleções inicializadas preguiçosamente e proxies. `Hibernate.initialize(cat)` irá forçar a inicialização de um proxy, `cat`, contanto que a `Session` esteja ainda aberta. `Hibernate.initialize( cat.getKittens() )` tem um efeito similar para a coleção de kittens.

Outra opção é manter a `Session` aberta até que todas as coleções e proxies necessários sejam carregados. Em algumas arquiteturas de aplicações, particularmente onde o código que acessa os dados usando Hibernate e o código que usa os dados estão em diferentes camadas da aplicação ou diferentes processos físicos, será um problema garantir que a `Session` esteja aberta quando uma coleção for inicializada. Existem dois caminhos básicos para lidar com esse problema:

- Em aplicações web, um filtro servlet pode ser usado para fechar a `Session` somente no final da requisição do usuário, já que a renderização da visão estará completa (o pattern *Open Session In View*). Claro, que isto cria a necessidade de um correto manuseio de exceções na infraestrutura de sua aplicação. É vitalmente importante que a `Session` esteja fechada e a transação terminada antes de retornar para o usuário, mesmo que uma exceção ocorra durante a renderização da view. Veja o Wiki do Hibernate para exemplos do pattern "Open Session In View"

- Em uma aplicação com uma camada de negócios separada, a lógica de negócios deve "preparar" todas as coleções que serão usadas pela camada web antes de retornar. Isto sgnifica que a camada de negócios deve carregar todos os dados e retorná-los já inicializados para a camada de apresentação. Usualmente a aplicação chama `Hibernate.initialize()` para cada coleção que será usada pela camada web (essa chamada de método deve ocorrer antes da sessão ser fechada ou retornar a coleção usando uma consulta Hibernate com uma cláusula `FETCH` ou um `FetchMode.JOIN` na `Criteria`. Fica muito mais fácil se você adotar o pattern *Command* ao invés do *Session Facade*.

- Você também pode anexar um objeto prevaimente carregado em uma nova `Session` `merge()` or `lock()` antes de acessar coleções não inicializadas (ou outros proxies). O Hibernate não faz e certamente `não deve` isso automaticamente pois isso introduziria semantica em transações ad hoc.

As vezes você não quer inicializar uma coleção muito grande, mas precisa de algumas informações (como o tamanho) ou alguns de seus dados.

Você pode usar um filtro de coleção para saber seu tamanho sem a inicializar:

```
( (Integer) s.createFilter( collection, "select count(*)" ).list().get(0) ).intValue()
```

O método `createFilter()` é usado também para retornar algus dados de uma coleção eficientemente sem precisar inicializar a coleção inteira:

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResults(10).list();
```

19.1.5. Usando busca em lote

O Hibernate pode fazer uso eficiente de busca em lote, isto é, o Hibernate pode carregar diversos proxies não inicializados se um proxy é acessado (ou coleções. A busca em lote é uma otimização da estratégia de select fetching). Existe duas maneiras em que você pode usar busca em lote: no nível da classe ou no nível da coleção.

A recuperação em lote para classes/entidades é mais fácil de entender. Imagine que você tem a seguinte situação em tempo de execução: Você tem 25 instâncias de `Cat` carregadas em uma `Session`, cada `Cat` tem uma referência ao seu `owner`, que é da classe `Person`. A classe `Person` é mapeada com um proxy, `lazy="true"`. Se você iterar sobre todos os Cat's e chamar `getOwner()` em cada, o Hibernate irá por padrão executar 25 comandos `SELECT()`, para buscar os proxies de owners. Você pode melhorar esse comportamento especificando um `batch-size` no mapeamento da classe `Person`:

```
<class name="Person" batch-size="10">...</class>
```

O Hibernate irá executar agora apenas três consultas, buscando por vez, 10, 10 e 5 Person.

Você também pode habilitar busca em lote de uma coleção. Por exemplo, se cada `Person` tem uma coleção lazy de `Cats`, e 10 pessoas já estão carregados em uma `Sesssion`, serão gerados 10 `SELECT`s ao se iterar todas as pessoas, um para cada chamada de `getCats()`.. Se você habilitar a busca em lote para a coleção de `cats` no mapeamento da classe `Person`, o Hibernate pode fazer uma pré carga das coleções:

```
<class name="Person">
    <set name="cats" batch-size="3">
        ...
    </set>
</class>
```

Com um `batch-size`de 8, o Hibernate irá carregar 3, 3, 3, 1 coleções em 4 `SELECT`s. Novamente, o valor do atributo depende do número esperado de coleções não inicialiadas em determinada `Session`.

A busca em lote de coleções é particularmente útil quando você tem uma árvore encadeada de items, ex. o típico padrão bill-of-materials (Se bem que um *conjunto encadeado* ou *caminho materializado* pode ser uma opção melhor para árvores com mais leitura)

19.1.6. Usando subselect fetching

Se uma coleção ou proxy simples precisa ser recuperado, o Hibernate carrega todos eles rodando novamente a query original em um subselect. Isso funciona da mesma maneira que busca em lote, sem carregar tanto.

19.1.7. Usando busca preguiçosa de propriedade

O Hibernate3 suporta a carga posterior de propriedades individuais. Essa técnica de otimização também conhecida como *fetch groups*. Veja que isso é mais uma funcionalidade de marketing já que na prática, é mais importante otimização nas leituras dos registros do que na leitura das colunas. Porém, carregar apenas algumas propriedades de uma classe pode ser útil em casos extremos, onde tabelas legadas podem ter centenas de colunas e o modelo de dados não pode ser melhorado.

Para habilitar a carga posterior de propriedade, é preciso setar o atributo `lazy` no seu mapeamento de propriedade:

```
<class name="Document">
       <id name="id">
        <generator class="native"/>
    </id>
    <property name="name" not-null="true" length="50"/>
    <property name="summary" not-null="true" length="200" lazy="true"/>
```

```
    <property name="text" not-null="true" length="2000" lazy="true"/>
</class>
```

A carga posterior de propriedades requer instrumentação de bytecode! Se suas classes persistentes não forem melhoradas, o Hibernate irá ignorar silenciosamente essa configuração e usará busca imediatamente.

Para instrumentação de bytecode, use a seguinte tarefa do Ant:

```
<target name="instrument" depends="compile">
    <taskdef name="instrument" classname="org.hibernate.tool.instrument.InstrumentTask">
        <classpath path="${jar.path}"/>
        <classpath path="${classes.dir}"/>
        <classpath refid="lib.class.path"/>
    </taskdef>

    <instrument verbose="true">
        <fileset dir="${testclasses.dir}/org/hibernate/auction/model">
            <include name="*.class"/>
        </fileset>
    </instrument>
</target>
```

Um modo diferente (melhor?) para evitar a leitura desnecessária de colunas, pelo menos para transações de somente de leitura é usar as características de projeção do HQL ou consultas por Critérios. Isto evita a necessidade de processamento de bytecode em tempo de execução e é certamente uma solução melhor.

Você pode forçar a carga antecipada da propriedades usando `fetch all properties` no HQL.

19.2. O Cache de segundo nível

Uma `Session` do Hibernate é um cache um de dados persistentes em nivel transacional. É possível configurar um cluster ou cache a nível da JVM (nível `SessionFactory`)em uma base classe-por-classe e coleção-por-coleção. Você pode até mesmo associar um cache a um cluster. Tenha cuidado. Os cache não estão a par das mudanças feitas nos dados persistentes por outras aplicações (embora eles possam ser configurados para expirar os dados de cached regularmente).

Você tem a opção para dizer ao Hibernate qual implementação de caching ele deve usar especificando o nome de uma classe que implementa a interface `org.hibernate.cache.CacheProvider` usando a propriedade `hibernate.cache.provider_class`. O Hibernate vem com várias implementações para integração com vários providerss de cache open-source (listados abaixo); Adicionalmente, você poderia criar sua próprio implementação como descrito acima. Veja que antes d versão de 3.2 era usado o EhCache como provider de cache padrão; isso não é mais o caso a partir de 3.2.

Tabela 19.1. Cache Providers

Cache	classe Provider	Tipo	Suporte a Cluster	Suporta consultas no Cache
Hashtable (não indicado para uso em produção)	`org.hibernate.cache.HashtableCacheProvider`	memoria		sim
EHCache	`org.hibernate.cache.EhCacheProvider`	memoria, disco		sim

Cache	classe Provider	Tipo	Suporte a Cluster	Suporta consultas no Cache
OSCache	`org.hibernate.cache.OSCacheProvider`	memoria, disco		sim
SwarmCache	`org.hibernate.cache.SwarmCacheProvider`	Clusterizado(ip multicast)	sim (invalidação de cluster)	
JBoss TreeCache	`org.hibernate.cache.TreeCacheProvider`	Clusterizado (ip multicast), transacional	sim (replicaçaão)	sim (clock sync req.)

19.2.1. Mapeamento do cache

O elemento `<cache>` do mapeamento de uma classe ou coleção tem a seguinte forma :

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"  (1)
    region="RegionName"                                              (2)
    include="all|non-lazy"                                           (3)
/>
```

(1) `usage` (obrigatório) especifica a estratégia de caching: `transactional`, `read-write`, `nonstrict-read-write` or `read-only`

(2) `region` (opicional, valor default o nome da classe da nome da role de collection) especifica o nome da região do caché de segundo nível

(3) `include` (opcional, valor default `all`) `non-lazy` especifica que as as propriedade de uma entidade mapeada com `lazy="true"` não serão cacheadas quando o atributo level lazy fetching estiver habilitado

Outra maneira (prefierivel), você pode especifcar os elementos `<class-cache>` e `<collection-cache>` no arquivo `hibernate.cfg.xml`.

O atributo `usage` especifica a *estratégia de concorrencia do cache*.

19.2.2. Strategia: somente leitura

Se sua aplicação precisar ler mas nunca modificar as intancias de classes persistentes, deve ser usado um cache `read-only`. Essa é a estratégia mais simples e de melhror performance. Inclusive é perfeitamente segura para o uso em cluster.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

19.2.3. Strategia: read/write

Em aplicações que precisam atualizar os dados, um cache `read-write` é o mais apropriado. Essa estratégia de cache não deve ser usada se o nivel de isolamento da transação requido for serializable. Se esse cache for usado em ambeinte JTA, você tem que dar um nomea para a estratégia especificando a propriedade `hiberna-`

te.transaction.manager_lookup_class para obtter uma TransactionManager JTA. Em outros ambientes, você deve assegurar que a transação foi completada quando Session.close() ou Session.disconnect() for chamado. Se você quiser usar essa estratégia em cluster, você deve assegurar que a implementaçã de cache utilizada suporta locking. Os caches que vem junto com o Hibernate *não* suportam.

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

19.2.4. Estratégia: nonstrict read/write

Se a aplicação apenas ocasionalmente precisa atualizar os dados (ie. se for extremamente improvável que duas transações tentem atualizar o mesmo item simultaneamente) e um rígido isolamento de transação não for requerido, um cache nonstrict-read-write pode ser apropriado. Se o cache for usado em um ambiente JTA, você tem que especificar hibernate.transaction.manager_lookup_class. Em outros ambientes, você deve assegurar que a transação esteja completa quando Session.close() ou Session.disconnect() for chamado.

19.2.5. Estratégia: transactional

A estratégia de cache transactional provê um completo suporte para provedores de cache transacionais como JBoss TreeCache. Tal cache só pode ser usado em um ambiente JTA e você tem que especificar hibernate.transaction.manager_lookup_class.

Nenhum dos provedores de cache suporta todas as estratégias de concorrencia de cache. A seguinte tabela mostra quais provedores são compatíveis com que estratégias de concorrencia.

Tabela 19.2. Suporte a estratégia de concorrencia em cache

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (não é projetado para uso em produção)	sim	sim	sim	
EHCache	sim	sim	sim	
OSCache	sim	sim	sim	
SwarmCache	sim	sim		
JBoss TreeCache	sim			sim

19.3. Administrando os caches

Sempre que você passa um objeto para save(), update() ou saveOrUpdate() e sempre que você recupera um objeto usando load(), get(), list(), iterate() ou scroll(), esse objeto é acrescentado no cache interno da Session..

Após isso, quando o `flush()` é chamado, o estado deste objeto será sincronizado com o banco de dados. Se você não quizer que esta sincronização aconteça ou se você estover processando um número enorme de objetos e precisa administrar memória de maneira eficaz, o método `evict()` pode ser usado para remover o objeto e suas coleções do cache de primeiro-nível.

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

A `Session` também provê o método `contains()` para determinar se uma instancia pertence ao cache da sessão.

Para remover completamente todos os objetos do cache da sessão, use `Session.clear()`

Para o cache de segundo-nível, há métodos definidos na `SessionFactory` parar remover o cache de estado de uma instancia, uma classe inteira, uma instancia de coleção ou uma coleção inteira.

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class);  //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections
```

O `CacheMode` controla como uma sessão em particular interage com o cache de segundo-nível.

- `CacheMode.NORMAL` - lê itens do e escreve itens no cache de segundo nível
- `CacheMode.GET` - lê itens do cache de segundo nível, mas não escreve no mesmo, exeto a atualização de dados
- `CacheMode.PUT` - escreve itens no cache de segundo nível, mas não lê dados do mesmo
- `CacheMode.REFRESH` - escreve item no cache de segundo nivel, mas não lê do mesmo anulando o efeito de `hibernate.cache.use_minimal_puts`, e forcando uma atualização do caché de segundo nível para todos os items carregados do bando de dados

Para navegar no conteudo do cache de segundo ou na regiação de consulta do cache, use a API `Statistics`:

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

Você precisará habilitar estatísticas, e, opicionalmente, força o Hibernate ra manter as entradas no caché um formato mais compreensível:

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

19.4. O caché de consultas

Os result sets de consultas também podem ser cacheados. Isto só é útil para consultas que freqüentemente são executadas com os mesmos parâmetros. Para usar o cache de consulta você tem que habilitá-lo primeiro:

```
hibernate.cache.use_query_cache true
```

Esta configuração causa a criação de duas regiões novas no caché - um contendo os result set de consultas (`org.hibernate.cache.StandardQueryCache`), o outra contendo timestamps das ultimas atualizações para tabelas consultaveis (org.hibernate.cache.UpdateTimestampsCache). Veja que o caché de consulta não coloca em caché o estado das entidades que estão no result set; ele coloca em caché apenas os valores dos identificadores e resultados de tipo de valor. Assim caché de consulta sempre deve sempre ser usado junto com o caché de segundo nível.

A maioria das consultas não se beneficia de tratamento de caché, assim por padrão as consultas não são tratadas no caché. Para habilitar o tratamento de caché para consultas,chame `Query.setCacheable(true)`. Esta chamada permite a consulta recuperar resultados existente no caché ou acrescentar seus resultados no caché quando for executada.

Se você quiser um controle bem granulado sobre aspolíticas de expiração do caché de consultas, você pode especificar uma região do caché nomeada para uma consulta emm particular chamando `Query.setCacheRegion()`.

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

Se a consulta deve forçar uma atualização de sua região no caché de consulta, você dever chamar `Query.setCacheMode(CacheMode.REFRESH)`. Isto é particularmente útil em casos onde os dados subjacentes podem ter sido atualizados por um processo separado (i.e., não modificados através do Hibernate) e permite a aplicação uma atualização seletiva dos result sets de uma consulta em particular. Esta é uma alternativa mais eficiente do que a limpeza de uma região do caché de consultas via `SessionFactory.evictQueries()`.

19.5. Entendendo a performance de coleções

Nós já gastamos algum tempo falando sobre coleções. Nesta seção nós veremos com mais detalhes um par mais assuntos sobre como coleções se comportam em tempo de execução.

19.5.1. Taxonomania

O Hibernate define três tipos básicos de coleções:

- coleções de valores
- associações um para muitas
- associações muitas para muitas

Esta classificação distingue as várias tabelas e foreing key mas não nos diz nada do que nós precisamos saber sobre o modelo de relational. Entender completamente estrutura relational e as características de desempenho, nós também temos que considerar a estrutura da chave primária que é usada pelo Hibernate para atualizar ou apagar linhas na coleção. Isto sugere a seguinte classificação :

- coleções indexadas
- sets

- bags

Todas as coleções indexadas (maps, lists, arrays) têm uma chave primária formadas pelas colunas `<key>` e `<index>`. Nestes casos as atualizações de coleção são extremamente eficientes - a chave primária pode ser indexada eficazmente e uma linha em particular pode ser localizada eficazmente quando o Hibernate tenta atualiza-la ou apaga-la.

Os conjuntos (sets) têm uma chave primária que consiste de `<key>` e colunas de elemento. Isto pode ser menos eficiente para alguns tipos de elementos de coleção, particularmente elementos compostos ou textos grandes ou campos binários; o banco de dados pode não ser capaz de indexar uma chave primária complexa de modo eficiente. Por outro lado, em associações um-para-muitos ou muitos-para-muitos, particularmente no caso de identificadores sintéticos, é provável que sejam eficientes. (Nota: se você quer que o `SchemaExport` crie de fato uma chave primária de um `<set>` , você tem que declarar todas as colunas como `not-null="true"`.)

Os mapeamentos `<idbag>` definem uma chave substituta, assim eles sempre são muito eficientes a atualização. Na realidade, esse é o melhor caso.

Os Bags são o pior caso. Considerando que um bag permite que valores de elementos duplicadose não tem nenhuma coluna ndice, nenhuma chave primária pode ser definida. O Hibernate não tem nenhum modo de distinguir linhas duplicadas. O Hibernate soluciona este problema removendo (em um único `DELETE`) completamente e recriando a coleção sempre que ele safre alguma mudanã. Isto pode ser muito ineficiente.

Veja que para uma associação um-para-muitos , a "chave primária" pode não ser a chave primária física da tabela do banco de dados - mas até mesmo neste caso, a classificação anterior ainda é útil. (Ainda reflete como o Hibernate "localiza" linhas individuais da coleção.)

19.5.2. Lists, maps, idbags e sets são as coleções mais eficientes de se atualizar

A partir da discussão anterior, deve ficar claro que coleções indexadas e sets (normalmente) permitem perações mais eficiente em termos de adição, remoção e atualização elementos.

Há, discutivelmente, mais uma vantagem que as coleções indexadas tem sobre sobre outros conjuntos para associações muitos-para-muitos ou coleções dos valores. Por causa da estrutura do `Set`, o Hibernate nem sequer atualiza uma linha com `UPDATE`, quando um elemento "é alterado". As mudanças em um `Set` sempre são feitas através de `INSERT` e `DELETE` (de linhs individuais). Mais uma vez, estas regras não se aplicam a associações um-para-muitos.

Depois de observar que os arrays não podem ser lazy, nós concluiríamos que lists, maps e idbags são mais performáticos para a maioria dos tipos coleções(não inversas), com os conjuntos (sets) vindo logo atrás. Espera-se que os sets sejam o o tipo mais comum de coleção nas aplicações Hibernate. Isto se deve ao fato de que a semântica do "set" é muito parecida com o modelo relational.

Sem dúvida, em modelos Hibernate de domínio bem definidos, nós normalmente vemos que a maioria das coleções são na realidade associações um-para-muitos com `inverse="true"`. Para estas associações, a atualização é controlada pelo lado da associação muitos-para-um, e as considerações de desempenho na atualização de coleção simplesmente não se aplicam.

19.5.3. Bags e lists são as coleções inversas mais eficientes

Antes que você condene os bag para sempre, há um caso particular no qual os bags (e também lists) são muito mais performáticos que os sets. Para uma coleção com `inverse="true"` (o idioma fr relacionamento padrão

um-para-muitos, por exemplo) nós podemos acrescentar elementos a uma bag ou list sem precisar inicializar (recupear) os elementos da bag! Isso se deveao fato que `Collection.add()` ou `Collection.addAll()` sempre têm que devolver true para uma bag ou `List` (ao contrário de `Set`). Isto pode fazer o seguinte código comum funcionar muito mais rápido.

```
Parent p = (Parent) sess.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);  //no need to fetch the collection!
sess.flush();
```

19.5.4. Deletando tudo de uma vez

Ocasionalmente, apagar os elementos de coleção um por um pode ser extremamente ineficiente. O Hibernate não é completamente estúpido, e sabe que não deve fazer isso, no caso de uma coleção ser lima (se você tiver executado `list.clear()`, por exemplo). Neste caso, o Hibernate executará um simples `DELETE` e pronto!

Suponha que nós acrescentamos um único elemento a uma coleção de tamanho vinte e então tentarmos remover dois elementos. O Hibernate emitirá uma declaração`INSERT` e duas declarações `DELETE` (a menos que a coleção seja uma bag). Isto é certamente o desejado.

Porém, suponha que se nós removermos dezoito elementos, deixando dois e então adicionarmos três elementos novos. Existem duas formas possíveis de se proceder

- deletar as dezoito linhas uma por uma eentão inserir as três linhas
- remover toda a coleção (em um SQL `DELETE`) e inserir todas os cinco elementos atuais (um por um)

O Hibernate não é inteligente o bastante para saber que a segunda opção provavelmente é a mais rápida neste caso. (E provavelmente seria indesejável que o Hibernate fosse tão inteligente; tais comportamentos poderiam confundir as trigger do banco de dados , etc.)

Felizmente, você pode forçar este comportamento (ie. a segunda estratégia) a qualquer momento descartando (ie. desreferenciando) a coleção original e criando uma nova coleção com todos os elementos atuais instanciados. Isto pode ser muito útil e poderoso de vez em quando .

Claro que, a deleção com apenas um comando não se apliqua a coleções mapeadas com `inverse="true"`.

19.6. Monitorando o desempenho

A Otimização não é tem muito uso sem o monitoramento e o acesso a números de desempenho. O Hibernate provê uma grande gama de informações sobre suas operações internas. Estatísticas no Hibernate estão disponívis através da `SessionFactory`.

19.6.1. Monitorando a SessionFactory

Você pode acessar as métricas da `SessionFactory` de dois modos. Sua primeira opção é chamar `sessionFactory.getStatistics()` e ler ou exibir as `Statistics` você mesmo.

O Hibernate também pode usar a JMX para publicar métrica se você habilitar o MBean `StatisticsService` . Você pode habilitar um único MBean para todos seu `SessionFactory` ou um por fábrica. Veja o código seguinte para exemplos de uma configuração mais detalhada:

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

TODO: Isto não faz sentido: No primeiro caso, nós recuperamos e usamos o MBean diretamente. No segundo, temos que passar o nome JNDI em que a fábrica de sessão está publicada antes de poder usar-la. Use `hibernateStatsBean.setSessionFactoryJNDIName("my/JNDI/Name")`

Você pode (des)ativar o monitorando para uma `SessionFactory`

- via configuração: set `hibernate.generate_statistics` para `false`
- em tempo de execução: `sf.getStatistics().setStatisticsEnabled(true)` ou `hibernateStatsBean.setStatisticsEnabled(true)`

As estatísticas podem ser resetadas via programação usando o método `clear()`. Um resumo pode ser enviado a um logger (nível info) usando o o método `logSummary()`.

19.6.2. Métricas

O Hibernate provê um grande número de métricas, de informações básicas até informações especializadas pertinente sómente a certos cenários. As métrics disponíveis são descritas na interface da API `Statistics` , em três categorias:

- Métricas relacionadas ao uso geral da `Session`, como o número de sessões abertas, conexões de JDBC recuperadas, etc.
- Métricas relacionadas as entidades, coleções, consultas , e cachés com um todo (também conhecidas como métrica global),
- Métrica detalhadas relacionadas a uma entidade em particular, coleção, consulta ou região do caché.

Por examplo, você pode verificar o acesso, as perdas e a media de entidades, coleções e consultas, e o tempo médio usados pelas consultas. Lembre-se que o número de milissegundos está sujeito ao arredondamento do Java. O Hibernate é amarrado à precisão de JVM, em algumas plataformas essa precisão pode ser limitada a precisão de 10 segundos.

São usados getters simples para acessar as métricas globais (i.e. não referentes a uma entidade particular, coleção, região do cache, etc.). Você pode acessar o métrica de uma entidade em particular, coleção ou região do caché através de seu nome, por representação HQL ou SQL para consultas. Por favor verifique as API do Java-

doc sobre `Statistics`, `EntityStatistics`, `CollectionStatistics`, `SecondLevelCacheStatistics`, e `QueryStatistics` para mais informaçções. Os código seguintes mostram um exemplo simples:

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times"  );
```

Para trabalhar com todas as entidades, coleções, consultas e regiões do caché, você pode recuperar a lista de nomes de entidades, coleções, e regiões do caché, com os seguintes métodos : `getQueries()`, `getEntityNames()`, `getCollectionRoleNames()`, e `getSecondLevelCacheRegionNames()`.

Capítulo 20. Guia de ferramentas

Engenharia reversa com Hibernate é possível usando um cojunto de plugins do Eclipse, ferramentas de linha de comando, e tasks Ant.

As *ferramentas Hibernate* atualmente incluem plugins para a IDE Eclipse assim como tasks Ant para a engenharia reversa de bancos de dados existentes:

- *Mapping Editor:*: Editor para arquivos de mapeamento Hibernate XML, com suporte a auto-completion e syntax highlighting. Também suporta auto-completion para nomes de classe e nomes de propriedade/campo, fazendo com que seja muito mais versátil que um editor de XML normal.
- *Console:*: O console é uma nova visão do Eclipse. Além de uma visão eme árvore de suas configurações de console, você adquire também uma visão interativa de suas classes persistentes e suas relações. O console lhe permite executar quueries HQL no seu banco de dados e ver o resultado diretamente no Eclipse.
- *Assistentes de desenvolvimento*: Vários wizards são providos com o Hibernate Eclipse tools; você pode usar um wizard para gerar rapidamente os arquivos de configuração do Hibernate(cfg.xml), ou você pode até mesmo fazer a engenharia reversa completa de um schema de banco de dados existente em classes POJO, arquivos de mapeamento do Hibernate. O wizard de engenharia reversa suporta templates customizaveis.
- *Tasks Ant :*

Por favor veja o pacote *Hibernate Tools* e sua documentação para mais informações.

Entretanto, o pacote principal do Hibernate vem com uma ferramenta integrada (que pode até mesmo ser usada "dentro" do Hibernate on-the-fly): *SchemaExport* chamado de `hbm2ddl`.

20.1. Geração automática de schema

A DDL pode ser gerada a partir de seus aruivos de mapeamento por um utilitario do Hibernate.O schema gerado inclui constraints de integridade (primary keys e foreing keys) para as entidades e tabelas de coleção. Também são criadas tabelas e sequences para campos sequenciais mapeados.

Você *precisa* especificar um `Dialeto` SQL através da propriedade `hibernate.dialect` ao usar esta ferramenta, a DDL é altamente específica ao forncedor de banco de dados.

Primeiro, customize seus arquivos de mapeamento para melhorar o schema gerado.

20.1.1. Personalizando o schema

Muitos elementos de mapeamento do Hibernate, definem atributos opcionais com os nomes `length`, `precision` and `scale`. Você pode setar o comprimento, a precisão e a escala de uma coluna com estes atributos.

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

Algumas tags também aceitam o atributo `not-null` (por gerar uma constraint de `NOT NULL` nas colunas da tabela) e um atributo `unique` (uma constraint `UNIQUE` em colunas da tabela).

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

O atributo `unique-key` pode ser usado para se agrupar colunas em ums simples constraint de chave unica. Atualmente, o valor do atributo `unique-key` é *not* usado no nome da constraint no DDL gerado, somente para agrupar as colunas no arquivo de mapeamento.

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
<property name="employeeId" unique-key="OrgEmployee"/>
```

O atributo `index` especifica o nome do ndice que será criado usando a coluna ou colunas mapeadas. Pode-se agrupar múltiplas colunas no mesmo índice, simplesmente especificando o mesmo nome de índice.

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

O atributo `foreign-key` pode ser usado para se sobrescrever o nome de qualquer constraint foreign key gerada.

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

Muitos elementos de mapeamento também aceitam uma elemento filho `<column>`. Isto é particularmente útil para se mapear tipos multi-coluna:

```
<property name="name" type="my.customtypes.Name"/>
    <column name="last" not-null="true" index="bar_idx" length="30"/>
    <column name="first" not-null="true" index="bar_idx" length="20"/>
    <column name="initial"/>
</property>
```

O atributo `default` permite especificar um valor default por uma coluna (você deve atribuir o mesmo valor à propriedade mapeada antes de salvar uma nova instancia da classe mapeada).

```
<property name="credits" type="integer" insert="false">
    <column name="credits" default="10"/>
</property>
```

```
<version name="version" type="integer" insert="false">
    <column name="version" default="0"/>
</property>
```

O atributo `sql-type` permite ao usuário anular o mapeamento default de um tipo do Hibernate para um tipo de dados SQL.

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property>
```

O atributo `check` permite especificar uma constraint de cheque.

```
<property name="foo" type="integer">
    <column name="foo" check="foo > 10"/>
</property>
```

```
<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class>
```

Tabela 20.1. Resumo

Atributo	Valores	Interpretação
`length`	número	tamanho da coluna
`precision`	número	precisão decimal da coluna
`scale`	número	escala decimal da coluna
`not-null`	`true|false`	especifica que a coluna possa ou não ter valor nulo
`unique`	`true|false`	especifica que a coluna possui uma constraint de unicidade
`index`	`index_name`	specifies especifica o nome de um índice (multi-coluna)
`unique-key`	`unique_key_name`	especifica o nome da constraint multi-columana de unicidade
`foreign-key`	`foreign_key_name`	especifica o nome da constraint foreign key gerada por uma associação, para um elemento mapeado como `<one-to-one>`, `<many-to-one>`, `<key>`, ou `<many-to-many>`. Veja aquele o lados `inverse="true"` não é considerado pelo `SchemaExport`.
`sql-type`	`Tipo SQL da coluna`	sobrescreve o tipo default da coluna (somente para elementos `<column>`)
`default`	Expressão SQL	especifiqua um valor default para a coluna
`check`	Expressão SQL	cria uma check constraint coluna ou tablea

O elemento `<comment>` permite especificar comentários a serem inseridos no schema gerado.

```
<class name="Customer" table="CurCust">
    <comment>Current customers only</comment>
    ...
</class>
```

```
<property name="balance">
    <column name="bal">
        <comment>Balance in USD</comment>
    </column>
</property>
```

Isto resulta em um `comentário na tabela` ou `comentário na coluna` no DDL gerado (se for suportado).

20.1.2. Rodando a ferramenta

A ferramenta `SchemaExport` gera um script DDL por padrão e/ou executa as declarações DDL.

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaExport` *options mapping_files*

Tabela 20.2. Opções de linha de comando do `SchemaExport`

Opção	Descrição
`--quiet`	não gerar a saida do script no device
`--drop`	apenas excluir as tabelas
`--create`	apenas criar as tabelas
`--text`	não executar o script no banco de dados
`--output=my_schema.ddl`	gerar o script ddl para o arquivo
`--naming=eg.MyNamingStrategy`	escolha a `NamingStrategy`
`--config=hibernate.cfg.xml`	carreguar as configurações do Hibernate de um arquivo de XML
`--properties=hibernate.properties`	carreguar as propriedades do banco de dados do arquivo
`--format`	formatar o SQL gerado de maneira legível
`--delimiter=;`	Configurar o delimitador de fim de linha para o script

Você pode até mesmo embutir o `SchemaExport` em sua aplicação:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

20.1.3. Propriedades

Podem ser especificadas propriedades de banco de dados

- como variaveis de ambiente com `-D`<*property*>
- no `hibernate.properties`
- em um arquivo de propriedades`--properties`

As propriedades obrigatórias são:

Tabela 20.3. Propriedades de conexão do SchemaExport

Nome da propriedade	Descrição
`hibernate.connection.driver_class`	classe do driver jdbc
`hibernate.connection.url`	url jdbc
`hibernate.connection.username`	usuário do banco de dados
`hibernate.connection.password`	password do usuário do banco de dados
`hibernate.dialect`	dialeto usado pelo hibernate

20.1.4. Usando o Ant

Você pode executar `SchemaExport` a partir de um script Ant:

```
<target name="schemaexport">
```

```
    <taskdef name="schemaexport"
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

20.1.5. Ataulizações Incrementais do schema

A ferramenta `SchemaUpdate` atualizará um schema existente com mudanças "incrementais." Veja que `SchemaUpdate` depende do suporte a metadata da API JDBC, assim sendo não funcionará com todos os drivers JDBC.

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaUpdate` *options mapping_files*

Tabela 20.4. Opções de linha de comando do `SchemaUpdate`

Opção	Descrição
`--quiet`	não gerar a saida no script no device
`--text`	não executar o script no banco de dados
`--naming=eg.MyNamingStrategy`	escolha a `NamingStrategy`
`--properties=hibernate.properties`	carreguar as propriedades do banco de dados do arquivo
`--config=hibernate.cfg.xml`	especificar o arquivo `.cfg.xml`

Você pode embutir o `SchemaUpdate` em sua aplicação:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

20.1.6. Usando o Ant para updates incrementais do schema

Você pode chamar `SchemaUpdate` a partir de um script Ant:

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
```

```
</target>
```

20.1.7. Validação do Schema

A ferramenta `SchemaValidator` validará se o schema de banco de dados existente "confere" com seus documentos de mapeamento. Veja que `SchemaValidator` depende do suporte a metadata da API JDBC, assim sendo não funcionará com todos os drivers JDBC. Esta ferramenta é extremamente útil para testes.

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaValidator` *options mapping_files*

Tabela 20.5. Opções de linha de comando do `SchemaValidator`

Opção	Descrição
`--naming=eg.MyNamingStrategy`	use o`NamingStrategy`
`--properties=hibernate.properties`	leia as propriedades do banco de daos do arquivo
`--config=hibernate.cfg.xml`	especifica o arquivo `.cfg.xml`

Você pode embutir o `SchemaValidator` em sua aplicação:

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

20.1.8. Usando o Ant para a validação de schema

Você pode chamar o `SchemaValidator` a partir de um script Ant:

```
<target name="schemavalidate">
    <taskdef name="schemavalidator"
        classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"
        classpathref="class.path"/>

    <schemavalidator
        properties="hibernate.properties">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

Capítulo 21. Exemplo: Mestre/Detalhe

Um das primeiras coisas que os novos usuários tentam fazer com o Hibernate é modelar uma relação do tipo mestre / detalhe. Há duas maneiras de se fazer isso. Por várias razões a maneira mais conveniente, especialmente para novos usuários, é modelar ambos `Parent` e `Child` como clases de entidades com uma associação `<one-to-many>` de `Parent` para `Child`. (A outra maneira é declarar `Parent` como um `<composite-element>`.) Agora, veja que a semântica dafault de uma associação mestre / detalhe (em Hibernate) se parece menos com a semântica habitual de uma relação mestre / detalhe do que o mapemento de elementos compostos. Nós explicaremos como usar uma *associação bidirectional um-para-muitos com cascateamento* para modelar uma relação de mestre / detalhe de maneira eficaz e elegante. Não é difícil!

21.1. Uma nota sobre coleções

As coleções do Hibernate são consideradas compo parte dá lógica da entidade que a possui; nunca das entidades contidas na coleção. Esta é uma distinção crucial! E tem as seguintes conseqüências:

- Quando nós removermos / adicionamos um objeto de / para uma coleção, o número de versão do owner da coleção é incrementado.
- Se um objeto que foi removida de uma coleção for uma instancoa de um tipo de valor (ex, um elemento composto), esse objeto deixará de ser persistente e seu estado será completamente removido do banco de dados. Da mesma maneira, adicionar uma instancia de um tipo de valor à coleção fará com que seu estado imediatamente passe a ser persistente.
- Por outro lado, se uma entidade for removida de uma coleção (uma associação um-para-muitos ou muitos-para-muitos), por padrão, ela não será apagada . Este comportamento é completamente consistente - uma mudança do estado interno de outra entidade não deveria fazer a entidade associada desaparecer! Igualmente, adicionar uma entidade a uma coleção não faz aquela entidade ficar persistente, por padrão.

Ao invés disso, o comportamento padrão ao se acrescentar uma entidade em uma coleção é criar uma associação entre as duas entidades, e ao se remover também se remove a associação. Isto é muito apropriado para todos os casos. Exceto no caso de uma relação mestre / detalhe onde o cliclo de vida do detalhe ligada ao cliclo de vida do mestre.

21.2. Um-para-muitos bidirectional

Suponha que nós começamos com uma simples associação `<one-to-many>` .

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Se nós fôssemos executassemo o seguinte código

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate emitiria dois comandos SQL:

- um `INSERT` para criar o registro para `c`
- uma `UPDATE` para criar uma associação de `p` para `c`

Isto não é somente ineficiente, mas também viola qualquer constraint `NOT NULL` da coluna `parent_id`. Nós podemos resolver a violação de constraint not nul setand `not-null="true"` no mapeamento da coleção:

```
<set name="children">
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set>
```

Porém, esta não é a solução indicada.

A causa subjacente deste comportametno é que a associção (a foreign key `parent_id`) de `p` para `c` não é considerada parte do estado do objeto `Child` e não é criada no `INSERT`. Assim a solução é fazer da associação parte do mapeamento de d `Child`.

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

(Nós também precisamos acrescentar uma propriedade `parent` à classe `Child`.)

Agora que a entidade `Child` está administrando o estado da associação, nós iremos configurar a coleção para não atualizar a associação. Nós usaremos o atributo `inverse`.

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

O código seguinte seria usado para adicionar um nov `Child`

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

E agora, somente um `INSERT` SQL será emitido!

Para oitimizar as coisas mais um pouco, nós poderíamos criar um método `addChild()` para `Parent`.

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

Agora, o código para adicionar um `Child` deve se parecer com isso:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

21.3. ciclo de vida em cascata

A chamada explícita de `save()` ainda está confusa. Nós iremos melhorar isso usando cascata.

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Isto simplifica o código acima para:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

Semelhantemente, nós não precisamos iterar os detalhes quando salvamos ou deletamos um Mestre. O seguinte codigo remove `p` e todos `childrens` do banco de dados.

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

Porém, este código

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

Não removerá `c` do banco de dados; irá apanas remover a associação para `p`(e causar uma violação da constraint `NOT NULL`, neste caso). Você precisa chamar explicitamente `delete()` de `Child`.

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

Agora, em nosso caso, `Child` realmente não pode existir sem um parent. Assim se nós removermos `Child` da coleção, nós realmente queremos que ele seja apagado. Para isto, nós temos que usar `cascade="all-delete-orphan"`.

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Nota: embora o mapeamento da coleção especifique `inverse="true"`, O cascateamento ainda é processado através de iteração dos elementos de coleção. Assim se você quiser que um objeto seja salvo, deletado ou atualizado através de cascateamento, você tem que acrescentar esse comportamento na coleção. Uma simples chamada a `setParent()` não é o suficiente.

21.4. Tratamento em Cascata e `unsaved-value`

Suponha nós carregamos um `Parent` em uma `Session`, fizermos algumas mudanças em uma ação UI e desejemos persistir estas mudanças em uma nova sessão chamando `update()`. O `Parent` conterá uma coleção de childen e, desde a atualização em cascata esteja habilitada, O Hibernate precisa saber quais childrens estão realmente instanticiadas e que representam linhas existentes no banco de dados. Assumir que `Parent` e `Child` têm propriedades identificadoras geradas do do tipo `Long`. O Hibernate usará o valor do identificador e os valores das propriedade version/timestamp para determinar qual das children são novas. (Veja Seção 10.7, "Detecção automática de estado"). *No Hibernate3, não é mais necessário especificar `unsaved-value` explicitamente.*

O seguinte código atualizará `parent` e `child` e irá insrir `newChild`.

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Bem, isso é muito bem para o caso de identificadores gerados, mas e no caso de identificadores setados e identificadores compostos? Isto é mais difícil, visto que o Hibernate não pode usar a propriedade do identificador para distinguir entre uma nova instancia de objeto (com um identificador setado pelo usuário) e um objeto carragdo anteriormente em uma sessão. Neste caso, ou Hibernate usará a propriedade timestamp ou propriedade version, ou talvez o cache de segundo-nível ou, no pior caso, o banco de dados, para ver se a linha existe.

21.5. Conclusão

Existe muita coisa para se digerir e a primeira vista pode parecer confuso. Porém, na prática, tudo funciona muito bem. A maioria das aplicações Hibernate usam o padrão mestre / detalhe em muitas situações.

Nós mencionamos uma alternativa no primeiro parágrafo. Nenhum dos assuntos anteriores existe no caso de mapeamentos `<composite-element>`, que têm a exata semântica de uma relação de mestre / detalhe. Infelizmente, há duas grandes limitações para o uso de classes de elemento compostos: elementos compostos podem não possuir coleções, e elas não deveriam ser detalhes de qualquer entidade diferente de seu mestre.

Capítulo 22. Exemplo: Aplicação Weblog

22.1. Classes persistentes

As classes persistentes representam um weblog, e um artigo enviado a um weblog. Eles são ser modelados como uma relação de mestre / detalhe comum, mas nós usaremos uma bag ordenada, em vez de um conjunto (set).

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
```

```
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

22.2. Mapeamentos Hibernate

Os mapeamentos XML devem ser bem simples.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">
```

```
    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

22.3. Código Hibernate

A classe seguinte demonstra alguns dos tipos de coisas que nós podemos fazer com estas classes, usando Hibernate.

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
```

```
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
```

```
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
```

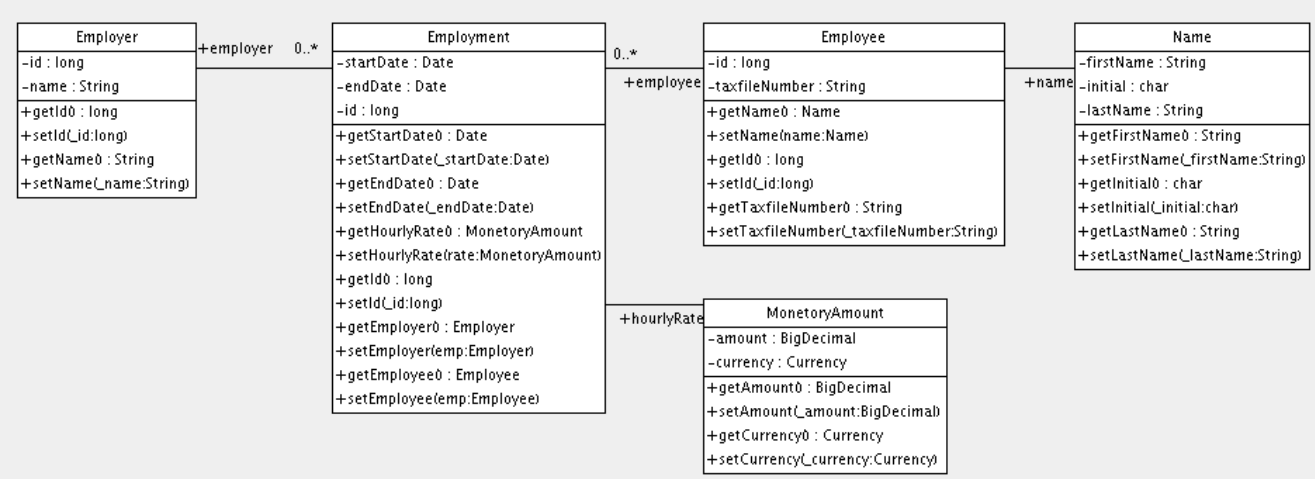
```
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.uniqueResult();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime > :minDate"
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

Capítulo 23. Exemplo: Vários Mapeamentos

Este capitulo mostra algums mapeamentos de associações mais complexos.

23.1. Employer/Employee

O modelo de seguinte relacionamento entre `Employer` e `Employee` utiliza uma entidade de classe atual (`Employment`) para representar a associação. Isto é feito porque pode-ser ter mais do que um período de trabalho para as duas partes envolvidas. Outros Componentes são usados para modelar valores monetários e os nomes do empregado.



Abaixo o código de um possível mapeamento:

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetaryAmount">
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
            </property>
            <property name="currency" length="12"/>
        </component>

        <many-to-one name="employer" column="employer_id" not-null="true"/>
        <many-to-one name="employee" column="employee_id" not-null="true"/>

    </class>

    <class name="Employee" table="employees">
```

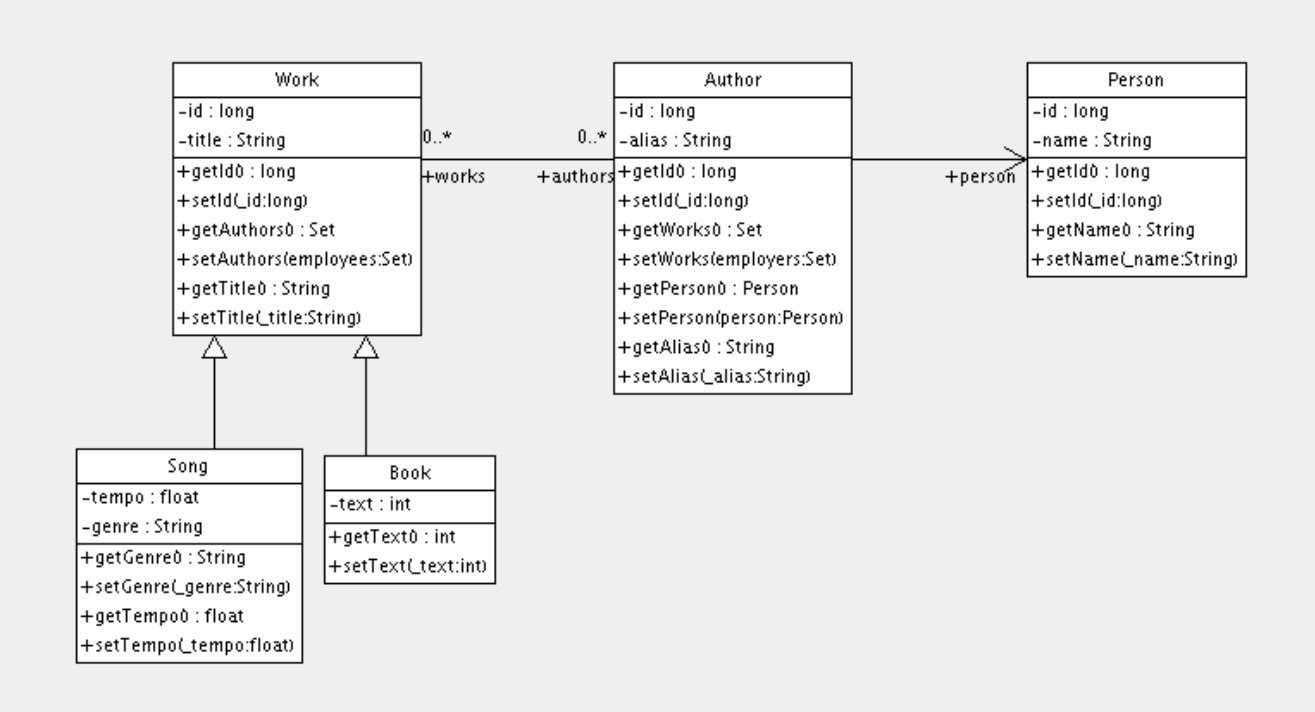
```
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

E abaixo o esquema da tabela gerado pelo `SchemaExport`.

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id) references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

23.2. Author/Work

Considere o seguinte modelo de relacionamento entre `Work`, `Author` e `Person`. Nós representamos o relacionamento entre `Work` e `Author` como uma associação muitos-para-muitos. Nós escolhemos representar o relacionamento entre `Author` e `Person` como uma associação um-para-um. Outra possibilidade seria ter `Author` extendendo `Person`.



O mapeamento do código seguinte representa corretamente estes relacionamentos:

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"/>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>

    </class>
```

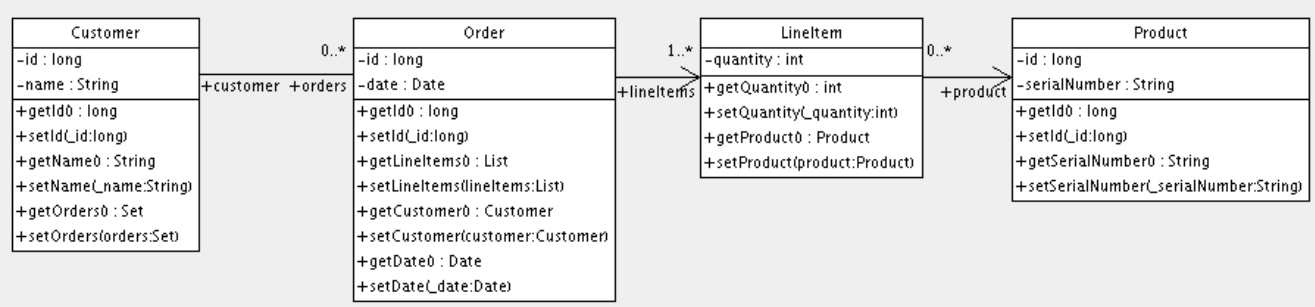
```
    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Existem quatro tabelas neste mapeamento. `works`, `authors` e `persons` recebem os dados de work, author e person, respectivamente. `author_work` é uma tabela de associação que liga authors à works. Abaixo o esquema das tabelas, gerados pelo `SchemaExport`.

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references works
```

23.3. Customer/Order/Product

Agora considere um modelo de relacionamento entre `Customer`, `Order` e `LineItem` e `Product`. Existe uma associação um-para-muitos entre `Customer` e `Order`, mas como devemos representar `Order` / `LineItem` / `Product`? Eu escolhi mapear LineItem como uma classe de associação representando a associação muitos-para-muitos entre `Order` e `Product`. No Hibernate, isto é conhecido como um elemento composto.



O código do mapeamento:

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items">
            <key column="order_id"/>
            <list-index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping>
```

`customers`, `orders`, `line_items` e `products` recebem os dados de customer, order, line_item e product, respectivamente. `line_items` também atua como uma tabela de associação ligando orders com products.

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
```

```
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders
```

23.4. Exemplos variados de mapeamento

Todos estes exemplos são retirados do conjunto de testes do Hibernate. Lá, você encontrará vários outros exemplos úteis de mapeamentos. Verifique o diretorio `test` da distribuição do Hibernate.

TODO: put words around this stuff

23.4.1. Associação um-para-um "Tipadas"

```
<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
            cascade="all">
        <formula>name</formula>
        <formula>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
            cascade="all">
        <formula>name</formula>
        <formula>'MAILING'</formula>
    </one-to-one>
</class>

<class name="Address" batch-size="2"
        check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
    <composite-id>
        <key-many-to-one name="person"
                column="personName"/>
        <key-property name="type"
                column="addressType"/>
    </composite-id>
    <property name="street" type="text"/>
    <property name="state"/>
    <property name="zip"/>
</class>
```

23.4.2. Exemplo de chave composta

```
<class name="Customer">

    <id name="customerId"
        length="10">
        <generator class="assigned"/>
```

```
    </id>

    <property name="name" not-null="true" length="100"/>
    <property name="address" not-null="true" length="200"/>

    <list name="orders"
            inverse="true"
            cascade="save-update">
        <key column="customerId"/>
        <index column="orderNumber"/>
        <one-to-many class="Order"/>
    </list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
            class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
            type="calendar_date"
            not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
            from LineItem li, Product p
            where li.productId = p.productId
                and li.customerId = customerId
                and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
            column="customerId"
            insert="false"
            update="false"
            not-null="true"/>

    <bag name="lineItems"
            fetch="join"
            inverse="true"
            cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>

</class>

<class name="LineItem">

    <composite-id name="id"
            class="LineItem$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
        <key-property name="productId" length="10"/>
    </composite-id>

    <property name="quantity"/>

    <many-to-one name="order"
            insert="false"
```

```
            update="false"
            not-null="true">
        <column name="customerId"/>
        <column name="orderNumber"/>
    </many-to-one>

    <many-to-one name="product"
            insert="false"
            update="false"
            not-null="true"
            column="productId"/>

</class>

<class name="Product">
    <synchronize table="LineItem"/>

    <id name="productId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="description"
        not-null="true"
        length="200"/>
    <property name="price" length="3"/>
    <property name="numberAvailable"/>

    <property name="numberOrdered">
        <formula>
            ( select sum(li.quantity)
            from LineItem li
            where li.productId = productId )
        </formula>
    </property>

</class>
```

23.4.3. Muitos-para-muitos com atributo de chave composta compartilhada

```
<class name="User" table="`User`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <set name="groups" table="UserGroup">
        <key>
            <column name="userName"/>
            <column name="org"/>
        </key>
        <many-to-many class="Group">
            <column name="groupName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>

<class name="Group" table="`Group`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <property name="description"/>
    <set name="users" table="UserGroup" inverse="true">
        <key>
            <column name="groupName"/>
            <column name="org"/>
        </key>
```

```
        <many-to-many class="User">
            <column name="userName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>
```

23.4.4. Conteúdo baseado em descriminação

```
<class name="Person"
    discriminator-value="P">

    <id name="id"
        column="person_id"
        unsaved-value="0">
        <generator class="native"/>
    </id>


    <discriminator
        type="character">
        <formula>
            case
                when title is not null then 'E'
                when salesperson is not null then 'C'
                else 'P'
            end
        </formula>
    </discriminator>

    <property name="name"
        not-null="true"
        length="80"/>

    <property name="sex"
        not-null="true"
        update="false"/>

    <component name="address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </component>

    <subclass name="Employee"
        discriminator-value="E">
            <property name="title"
                length="20"/>
            <property name="salary"/>
            <many-to-one name="manager"/>
    </subclass>

    <subclass name="Customer"
        discriminator-value="C">
            <property name="comments"/>
            <many-to-one name="salesperson"/>
    </subclass>

</class>
```

23.4.5. Associações em chaves alternativas

```
<class name="Person">

    <id name="id">
        <generator class="hilo"/>
```

```
    </id>

    <property name="name" length="100"/>

    <one-to-one name="address"
        property-ref="person"
        cascade="all"
        fetch="join"/>

    <set name="accounts"
        inverse="true">
        <key column="userId"
            property-ref="userId"/>
        <one-to-many class="Account"/>
    </set>

    <property name="userId" length="8"/>

</class>

<class name="Address">

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

    <property name="address" length="300"/>
    <property name="zip" length="5"/>
    <property name="country" length="25"/>
    <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
    <id name="accountId" length="32">
        <generator class="uuid"/>
    </id>

    <many-to-one name="user"
        column="userId"
        property-ref="userId"/>

    <property name="type" not-null="true"/>

</class>
```

Capítulo 24. Boas práticas

Escreva classes compactas e mapeie-as usando `<component>`.

Use uma classe `Endereco` para encapsular `rua`, `bairro`, `estado`, `CEP`. Isto promove a reutilização de código e simplifica o refactoring.

Declare propriedades identificadoras em classes persistentes.

O Hibernate constrói propriedades identificadoras opcionais. Existem todos os tipos de razões que explicam porquê você deveria utilizá-las. Nós recomendamos que os identificadores sejam 'sintéticos' (gerados, sem significado para o negocio).

Identifique chaves naturais.

Identifique chaves naturais para todas as entidades, e mapeie-as usando `<natural-id>`. Implemente `equals()` e `hashCode()` para comparar as propriedades que compõem a chave natural.

Coloque cada classe de mapeamento em seu próprio arquivo.

Não use um único código de mapeamento monolítico. Mapeie `com.eg.Foo` no arquivo `com/eg/Foo.hbm.xml`. Isto promove particularmente o bom senso no time de desenvolvimento.

Carregue os mapeamentos como recursos.

Faça o deploy dos mapeamentos junto com as classes que eles mapeiam.

Considere externalizar as strings de consultas.

Esta é uma boa prática se suas consultas chamam funções SQL que não sejam ANSI. Externalizar as strings de consultas para mapear arquivos irão tornar a aplicação mais portável.

Use bind de variaveis.

Assim como em JDBC, sempre substitua valores não constantes por "?". Nunca use a manipulação de strings para concatenar valores não constantes em uma consulta! Melhor ainda, considere usar parâmetros nomeados nas consultas.

Não gerencie suas conexões JDBC.

O Hibernate permite que a aplicação gerencie conexões JDBC. Esta abordagem deve ser considerada um último recurso. Se você não pode usar os provedores de conexão embutidos, considere fazer sua implementação a partir de `org.hibernate.connection.ConnectionProvider`.

Considere usar tipos customizados.

Suponha que você tenha um tipo Java, de alguma biblioteca, que precisa ser persistido mas não provê os acessórios necessários para mapeá-lo como um componente. Você deve implementar `org.hibernate.UserType`. Esta abordagem livra o código da aplicação de implementar transformações de/para o tipo Hibernate.

Use código manual JDBC nos gargalos.

Nas áreas de desempenho crítico do sistema, alguns tipos de operações podem se beneficiar do uso direto do JDBC. Mas por favor, espere até você *saber* se é um gargalo. E não suponha que o uso direto do JDBC é necessariamente mais rápido. Se você precisar usar diretamente o JDBC, vale a pena abrir uma `Session` do Hibernate e usar uma conexão JDBC. De modo que você possa ainda usar a mesma estratégia de transação e ocultar o provedor a conexão

Entenda o `Session` flushing.

De tempos em tempos a sessão sincroniza seu estado persistente com o banco de dados. O desempenho será afetado se este processo ocorrer frequentemente. Você pode algumas vezes minimizar o fluxo desnecessário desabilitando o fluxo automático ou até mesmo mudando a ordem das consultas e outras operações em

uma transação particular.

Em uma arquitetura de três camadas, considere o uso de objetos separados.

Ao usar uma arquitetura servlet / session bean , você poderia passar os objetos persistentes carregados dentro do session bean e da camada do servlet/JSP. Use uma nova sessão para cada pedido. Use `Session.merge()` ou `Session.saveOrUpdate()` para sincronizar objetos com o banco de dados.

Em uma arquitetura de duas camadas, considere usar contextos de persistência longos.

Transações de banco de dados têm que ser as mais curtas quanto possível para uma melhor escalabilidade. Porém, freqüentemente é neccessario implementar *transações de aplicação* com tempo longo de execução, um única unidade-de-trabalho do ponto de vista do usuário. Uma transação de aplicação pode atravessar várias ciclos de pedido / resposta do cliente. É comum o uso de objetos dessatachados para implementar transações de aplicação. Uma alternativa, extremamente apropriada em arquitetura de duas camadas, é manter um único contato de persistência aberto (sessão) para o ciclo de vida inteiro da transação de aplicação e simplesmente desconectar da conexão JDBC ao término de cada solicitação e reconectar no começo do solicitação subseqüente. Nunca compartilhe uma única sessão por mais de uma transação de aplicação, ou você estará trabalhando com dados desatualizados.

Don'Não trate exceções como recuperável.

Isto é mais uma prática necessária que uma "boa" prática. Quando uma exceção ocorrer, ocorre também um rollback na `Transaction` e a `Session` e fechada. Se isso não for feito, o Hibernate não pode garantir que o estado em memória representa com precisão o estado persistente. Em algum caso em especial, não use `Session.load()` para determinar se uma instancia com o determinado identificador existe no banco de dados; use `Session.get()` ou uma consulta.

De preferencia a carga tardia das associações.

Use o minimo possivel a recuperação antecipada. Use proxies e coleções lazy para a maioria das associações de classes que provávelmente não serão mantidas no cache segundo-nível. Para associações para classes no cache onde existe uma probabilidade de acesso extremamente alta, desabilite explicitamente recuperação antecipada usando `lazy="false"`. Quando for apropriado a recuperação por união (join fetching) para um caso em particular, use uma consulta com `left join fetch`.

Use o pattern *open session in view* uma *assembly phase* disciplinada para evitar problemas com dados não recuperados

O Hibernate livra o desenvolvedor de escrever os tediosos *Objetos de Transferência de Dados* (DTO). Em uma arquitetura de EJB tradicional, DTOs servem para dois propósitos: primeiro, eles resolvem o problema dos entity beans não serem serializaveis; segundo, eles implicitamente cuida da fase onde todos o dados que serão usados pela visão são recuperados e transformados em DTOs antes de se retornar o fluxo a camada e apresentação. O Hibernate elimina o primeiro propósito. Porém, você ainda precisará uma fase de conversão (pense em seus métodos de negocio como tendo um contrato rígido com a camada de apresentação sobre que dados estarão disponível nos objetos desassociados) a menos que você esteja preparado para ter o contexto de persistência (a sessão) aberto pelo processo de renderisação. Esta não é uma limitação do Hibernate! É uma exigência fundamental para acesso seguro de dados transacional .

Considere abstrair a sua logica de negocios do Hibernate.

Isole o código de acesso a dados(Hibernate) através de uma interface. Combine o uso dos padrões *DAO* e *Thread Local Session*. Você pode até mesmo ter algumas classes persistidas através de código JDBC, associado ao Hibernate através de um `UserType`. (Este conselho é para aplicações "suficientemente grandes"; não é apropriado para uma aplicação com cinco mesas!)

Não use mapeamento de associação exóticas.

São raros os casos reais de uso de associações muitos-para-muitos. A maioria do tempo você precisa de informação adicional armazenada em "tabelas associativas." Neste caso, é muito melhor para usar duass associações um-para-muitos para uma classe de associação intermediária. Na realidade, nós pensamos que a

maioria das associações é um-para-muitos e muitos-para-um, você deveria ter cuidadoso ao usar qualquer outro tipo de associação e se perguntar se ela é realmente neccessária.

De preferencia a associações bidirectionais.

É mais dificil fazer consultas em associações unidirectionais. Em uma aplicação grande, todas as associações devem ser navegáveis em ambas as direções nas consultas.