HIBERNATE - Persistência Relacional para Java Idiomático

1

Documentação de Referência Hibernate

3.6.0.Final

por Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, Steve Ebersole, e Hardy Ferentschik

and thanks to James Cobb (Graphic Design), Cheyenne Weaver (Graphic Design), Alvaro Netto, Anderson Braulio, Daniel Vieira Costa, Francisco gamarra, Gamarra, Luiz Carlos Rodrigues, Marcel Castelo, Paulo César, Pablo L. de Miranda, Renato Deggau, Rogério Araújo, e Wanderson Siqueira

Prefácio

Working with both Object-Oriented software and Relational Databases can be cumbersome and time consuming. Development costs are significantly higher due to a paradigm mismatch between how data is represented in objects versus relational databases. Hibernate is an Object/Relational Mapping solution for Java environments. The term Object/Relational Mapping refers to the technique of mapping data from an object model representation to a relational data model representation (and visa versa). See *http://en.wikipedia.org/wiki/Object-relational_mapping* for a good high-level discussion.

Nota

While having a strong background in SQL is not required to use Hibernate, having a basic understanding of the concepts can greatly help you understand Hibernate more fully and quickly. Probably the single best background is an understanding of data modeling principles. You might want to consider these resources as a good starting point:

- *http://www.agiledata.org/essays/dataModeling101.html*
- *http://en.wikipedia.org/wiki/Data_modeling*

Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities. It can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC. Hibernate's design goal is to relieve the developer from 95% of common data persistence-related programming tasks by eliminating the need for manual, hand-crafted data processing using SQL and JDBC. However, unlike many other persistence solutions, Hibernate does not hide the power of SQL from you and guarantees that your investment in relational technology and knowledge is as valid as always.

Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier. However, Hibernate can certainly help you to remove or encapsulate vendor-specific SQL code and will help with the common task of result set translation from a tabular representation to a graph of objects.

Por favor siga os seguintes passos, caso você seja inexperiente com o Hibernate, Mapeamento Objeto/Relacional ou mesmo Java:

1. Read *Capítulo 1, Tutorial* for a tutorial with step-by-step instructions. The source code for the tutorial is included in the distribution in the `doc/reference/tutorial/` directory.
2. Read *Capítulo 2, Arquitetura* to understand the environments where Hibernate can be used.

3. Verifique no diretório `eg/` em sua distribuição de Hibernate, do qual possui uma simples aplicação autônoma. Copie seu driver JDBC para o diretório `lib/` e edite `eg/hibernate.properties`, especificando valores corretos para o seu banco de dados. No diretório de distribuição sob o comando aviso, digite `ant eg` (usando Ant), ou sob Windows, digite `build eg`.

4. Use this reference documentation as your primary source of information. Consider reading [*JPwH*] if you need more help with application design, or if you prefer a step-by-step tutorial. Also visit *http://caveatemptor.hibernate.org* and download the example application from [*JPwH*].

5. As respostas das perguntas mais freqüentes podem ser encontradas no website Hibernate.

6. A terceira parte de demonstração, exemplos e tutoriais estão vinculadas no website Hibernate.

7. A Área de Comunidade no website Hibernate é um bom recurso para parceiros de design e várias soluções integradas. (Tomcat, JBoss AS, Struts, EJB, etc.)

There are a number of ways to become involved in the Hibernate community, including

- Trying stuff out and reporting bugs. See *http://hibernate.org/issuetracker.html* details.
- Trying your hand at fixing some bugs or implementing enhancements. Again, see *http://hibernate.org/issuetracker.html* details.
- *http://hibernate.org/community.html* list a few ways to engage in the community.
  - There are forums for users to ask questions and receive help from the community.
  - There are also *IRC* [http://en.wikipedia.org/wiki/Internet_Relay_Chat] channels for both user and developer discussions.
- Helping improve or translate this documentation. Contact us on the developer mailing list if you have interest.
- Evangelizing Hibernate within your organization.

Tutorial

Intencionado para novos usuários, este capítulo fornece uma introdução detalhada do Hibernate, começando com um aplicativo simples usando um banco de dados em memória. O tutorial é baseado num tutorial anterior desenvolvido por Michael Gloegl. Todo o código está contido no diretório `tutorials/web` da fonte do projeto.

Importante

Este tutorial espera que o usuário tenha conhecimento de ambos Java e SQL. Caso você tenha um conhecimento limitado do JAVA ou SQL, é recomendado que você inicie com uma boa introdução àquela tecnologia, antes de tentar entender o Hibernate.

Nota

Esta distribuição contém outro aplicativo de amostra sob o diretório de fonte do projeto `tutorial/eg`.

1.1. Parte 1 – A primeira aplicação Hibernate

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

Nota

Mesmo que usando qualquer banco de dados do qual você se sinta confortável, nós usaremos *HSQLDB* [http://hsqldb.org/] (o em memória, banco de dados Java) para evitar a descrição de instalação/configuração de quaisquer servidores do banco de dados.

1.1.1. Configuração

O primeiro passo em que precisamos tomar é configurar o ambiente de desenvolvimento. Nós usaremos o "layout padrão" suportado por muitas ferramentas de construção, tais como *Maven* [http://maven.org]. Maven, em particular, possui um excelente recurso de descrição disto *layout* [http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html]. Assim como este tutorial deve ser um aplicativo da web, nós criaremos e faremos uso dos diretórios `src/main/java`, `src/main/resources` e `src/main/webapp`.

Nós usaremos Maven neste tutorial, tirando vantagem destas capacidades de dependência transitiva assim como a habilidade de muitos IDEs de configurar automaticamente um projeto baseado no descritor maven.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.hibernate.tutorials</groupId>
    <artifactId>hibernate-tutorial</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>First Hibernate Tutorial</name>

    <build>
         <!-- we dont want the version to be part of the generated war file name -->
         <finalName>${artifactId}</finalName>
    </build>

    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
        </dependency>

        <!-- Because this is a web app, we also have a dependency on the servlet api. -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
        </dependency>

        <!-- Hibernate uses slf4j for logging, for our purposes here use the simple backend -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
        </dependency>

        <!-- Hibernate gives you a choice of bytecode providers between cglib and javassist -->
        <dependency>
            <groupId>javassist</groupId>
            <artifactId>javassist</artifactId>
        </dependency>
    </dependencies>

</project>
```

Dica

It is not a requirement to use Maven. If you wish to use something else to build this tutorial (such as Ant), the layout will remain the same. The only change is that you will need to manually account for all the needed dependencies. If you

use something like *Ivy* [http://ant.apache.org/ivy/] providing transitive dependency management you would still use the dependencies mentioned below. Otherwise, you'd need to grab *all* dependencies, both explicit and transitive, and add them to the project's classpath. If working from the Hibernate distribution bundle, this would mean `hibernate3.jar`, all artifacts in the `lib/required` directory and all files from either the `lib/bytecode/cglib` or `lib/bytecode/javassist` directory; additionally you will need both the servlet-api jar and one of the slf4j logging backends.

Salve este arquivo como `pom.xml` no diretório raiz do projeto.

1.1.2. A primeira Classe

Agora, iremos criar uma classe que representa o evento que queremos armazenar na base de dados. Isto é uma classe JavaBean simples com algumas propriedades:

```
package org.hibernate.tutorial.domain;

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 para o nome convencional dos métodos de propriedade getter e setter, como também a visibilidade privada dos campos. Este é um padrão de projeto recomendado, mas não requerido. O Hibernate pode também acessar campos diretamente, o benefício para os métodos de acesso é a robustez para o refactoring.

A propriedade `id` mantém um único valor de identificação para um evento particular. Todas as classes persistentes da 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, especialmente. aplicações web, precisam distinguir os objetos pelo identificador. Portanto, você deverá considerar esta, uma característica ao invés de uma limitação. Porém, nós normalmente não manipulamos a identidade de um objeto, conseqüentemente o método setter deverá ser privado. O Hibernate somente nomeará os identificadores quando um objeto for salvo. O Hibernate pode acessar métodos públicos, privados, e protegidos, como também campos públicos, privados, protegidos diretamente. A escolha é sua e você pode adaptar seu projeto de aplicação.

O construtor sem argumentos é um requerimento para todas as classes persistentes; O Hibernate precisa criar para você os objetos usando Java Reflection. O construtor pode ser privado, porém, a visibilidade do pacote é requerida para a procuração da geração em tempo de execução e recuperação eficiente dos dados sem a instrumentação de bytecode.

Salve este arquivo no diretório `src/main/java/org/hibernate/tutorial/domain`.

1.1.3. O mapeamento do arquivo

O Hibernate precisa saber como carregar e armazenar objetos da classe de persistência. É aqui que o mapeamento do arquivo do Hibernate entrará em jogo. O arquivo mapeado 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://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.domain">
[...]
</hibernate-mapping
>
```

Note que o Hibernate DTD é muito sofisticado. Você pode usar isso para auto-conclusão no mapeamento XML dos elementos e funções no seu editor ou IDE. Você também pode abrir o arquivo DTD no seu editor. Esta é a maneira mais fácil de ter uma visão geral de todos os elementos e funções e dos padrões, como também alguns comentários. Note que o Hibernate não irá carregar o arquivo DTD da web, e sim da classpath da aplicação. O arquivo DTD está

incluído no `hibernate-core.jar` (como também no `hibernate3.jar`, caso usando a vinculação de distribuição.

Importante

Nós omitiremos a declaração do DTD nos exemplos futuros 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 sobre as classes que não são classes-primárias de entidades) necessitam do tal mapeamento, para uma tabela na base de dados SQL:

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

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

    </class>

</hibernate-mapping>
```

Até agora, informamos o Hibernate sobre como fazer para persistir e carregar objetos da classe `Event` da tabela `EVENTS`, cada instância representada por uma coluna na tabela. Agora, continuaremos com o mapeamento de uma única propriedade identificadora para as chaves primárias da tabela. Além disso, como não precisamos nos preocupar em manipular este identificador, iremos configurar uma estratégia de geração de id's do Hibernate para uma coluna de chave primária substituta:

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

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

</hibernate-mapping>
```

O elemento `id` é a declaração de uma propriedade do identificador. O atributo do mapeamento `name="id"` declara que o nome da propriedade JavaBeans e informa o Hibernate a utilizar os métodos `getId()` and `setId()` para acessar a propriedade. A atributo da coluna informa o Hibernate qual coluna da tabela `EVENTS` mantém o valor de chave primária.

O elemento `generator` aninhado especifica a estratégia da geração do identificador (como os valores do identificador são gerados?). Neste caso, nós escolhemos `native`, do qual oferece

um nível de portabilidade dependendo no dialeto do banco de dados configurado. O Hibernate suporta o banco de dados gerado, globalmente único, assim como a aplicação determinada, identificadores. A geração do valor do identificador é também um dos muitos pontos de extensão do Hibernate e você pode realizar o plugin na sua própria estratégia.

Dica

`native` is no longer consider the best strategy in terms of portability. for further discussion, see *Seção 28.4, "Geração do identificador"*

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

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="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>
```

Assim como com o elemento `id`, a função `name` do elemento `property` informa ao Hibernate qual método getter e setter deverá usar. Assim, neste caso, o Hibernate irá procurar pelos métodos `getDate()`, `setDate()`, `getTitle()` e `setTitle()`.

Nota

Porque fazer o mapeamento da propriedade `date` incluído na função `column`, e no `title` não fazer? Sem a função `column` o Hibernate, por padrão, utiliza o nome da propriedade como o nome da coluna. Isto funciona bem para o `title`. Entretanto, o `date` é uma palavra-chave reservada na maioria dos bancos de dados, por isso seria melhor mapeá-lo com um nome diferente.

O mapeamento do `title` também não possui a função `type`. O tipo que declaramos e utilizamos nos arquivos mapeados, não são como você esperava, ou seja, funções de dados Java. Eles também não são como os tipos de base 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 irá tentar determinar a conversão correta e mapeará o `type` próprio, caso o tipo da função não estiver presente no mapeamento. Em alguns casos, esta detecção automática (que usa Reflection sobre as classes

Java) poderá não ter o padrão que você espera ou necessita. Este é o caso com a propriedade `date`. O Hibernate não sabe se a propriedade, que é do `java.util.Date`, pode mapear para uma coluna do tipo `date` do SQL, `timestamp` ou `time`. Nós preservamos as informações sobre datas e horas pelo mapeamento da propriedade com um conversor `timestamp`.

Dica

O Hibernate realiza esta determinação de tipo de mapeamento usando a reflexão quando os arquivos de mapeamentos são processados. Isto pode levar tempo e recursos, portanto se você inicializar o desempenho, será importante que você considere claramente a definição do tipo para uso.

Salve este arquivo de mapeamento como `src/main/resources/org/hibernate/tutorial/domain/Event.hbm.xml`.

1.1.4. Configuração do Hibernate

Nestas alturas, você deve possuir a classe persistente e seu arquivo de mapeamento prontos. É o momento de configurar o Hibernate. Primeiro, vamos configurar o HSQLDB para rodar no "modo do servidor".

Nota

Nós realizamos isto para que aqueles dados permaneçam entre as execuções.

Nós utilizaremos o Maven exec plugin para lançar o servidor HSQLDB pela execução: `mvn exec:java -Dexec.mainClass="org.hsqldb.Server" -Dexec.args="-database.0 file:target/data/tutorial"`. Você pode ver ele iniciando e vinculando ao soquete TCP/IP, aqui será onde nossa aplicação irá se conectar depois. Se você deseja iniciar uma nova base de dados durante este tutorial, finalize o HSQLDB, delete todos os arquivos no diretório `target/data`, e inicie o HSQLBD novamente.

O Hibernate conectará ao banco de dados no lugar de sua aplicação, portanto ele precisará saber como obter as conexões. Para este tutorial nós usaremos um pool de conexão autônomo (ao invés de `javax.sql.DataSource`). O Hibernate vem com o suporte para dois terços dos pools de conexão JDBC de código aberto: *c3p0* [https://sourceforge.net/projects/c3p0] e *proxool* [http://proxool.sourceforge.net/]. No entanto, nós usaremos o pool de conexão interna do Hibernate para este tutorial.

Cuidado

O pool de conexão interna do Hibernate não é recomendado para uso de produção. Ele possui deficiência em diversos recursos encontrados em qualquer pool de conexão apropriado.

Para as configurações do Hibernate, nós podemos usar um arquivo simples `hibernate.properties`, um arquivo mais sofisticado `hibernate.cfg.xml` ou até mesmo uma instalação programática completa. A maioria dos usuários prefere 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://www.hibernate.org/dtd/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"
>update</property>
```

```
        <mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration
>
```

Nota

Perceba que este arquivo de configuração especifica um DTD diferente

Configure a `SessionFactory` do Hibernate. A SessionFactory é uma fábrica global responsável por uma base de dados particular. Se você tiver diversas bases de dados, use diversas configurações `<session-factory>`, geralmente em diversos arquivos de configuração, para uma inicialização mais fácil.

Os primeiros quatro elementos `property` contêm a configuração necessária para a conexão JBDC. O elemento `property` do dialeto especifica a variante do SQL particular que o Hibernate gera.

Dica

In most cases, Hibernate is able to properly determine which dialect to use. See *Seção 28.3, "Resolução do Dialeto"* for more information.

O gerenciamento automático de sessão do Hibernate para contextos de persistência é bastante útil neste contexto. A opção `hbm2ddl.auto` habilita a geração automática de esquemas da base de dados, diretamente na base de dados. Isto também pode ser naturalmente desligado apenas removendo a opção de configuração ou redirecionado para um arquivo com ajuda do `SchemaExport` na tarefa do Ant. Finalmente, iremos adicionar os arquivos das classes de persistência mapeadas na configuração.

Salve este arquivo como `hibernate.cfg.xml` no diretório `src/main/resources`.

1.1.5. Construindo com o Maven

Nós iremos construir agora o tutorial com Maven. Você necessitará que o Maven esteja instalado; ele está disponível a partir do *Maven download page* [http://maven.apache.org/download.html]. O Maven gravará o arquivo `/pom.xml` que criamos anteriormente, além de saber como executar algumas tarefas do projeto básico. Primeiro, vamos rodar o objetivo `compile` para nos certificarmos de que tudo foi compilado até agora:

```
[hibernateTutorial]$ mvn compile
[INFO] Scanning for projects...
```

```
[INFO] ------------------------------------------------------------------------
[INFO] Building First Hibernate Tutorial
[INFO]    task-segment: [compile]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to /home/steve/projects/sandbox/hibernateTutorial/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Jun 09 12:25:25 CDT 2009
[INFO] Final Memory: 5M/547M
[INFO] ------------------------------------------------------------------------
```

1.1.6. Inicialização e Auxiliares

É hora de carregar e armazenar alguns objetos `Event`, mas primeiro nós temos de completar a instalação com algum código de infraestrutura. Você precisa inicializar o Hibernate pela construção de um objeto `org.hibernate.SessionFactory` global e o armazenamento dele em algum lugar de fácil acesso para o código da aplicação. O `org.hibernate.SessionFactory` é usado para obter instâncias `org.hibernate.Session`. O `org.hibernate.Session` representa uma unidade de single-threaded de trabalho. O `org.hibernate.SessionFactory` é um objeto global thread-safe, instanciado uma vez.

Criaremos uma classe de ajuda `HibernateUtil`, que cuida da inicialização e faz acesso a uma `org.hibernate.SessionFactory` mais conveniente.

```
package org.hibernate.tutorial.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return 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;
    }
```

```
}
```

Salve este código como `src/main/java/org/hibernate/tutorial/util/HibernateUtil.java`

Esta classe não só produz uma referência `org.hibernate.SessionFactory` global em seu inicializador estático, mas também esconde o fato de que utiliza um autônomo estático. Nós poderemos buscar pela referência `org.hibernate.SessionFactory` a partir do JNDI no servidor da aplicação ou qualquer outra localização para este assunto.

Se você der um nome à `SessionFactory` em seu arquivo de configuração, o Hibernate irá, de fato, tentar vinculá-lo ao JNDI sob aquele nome, depois que estiver construído. Outra opção melhor seria usar a implementação JMX e deixar o recipiente JMX capaz, instanciar e vincular um `HibernateService` ao JNDI. Essas opções avançadas são discutidas no documento de referência do Hibernate. Tais opções avançadas serão discutidas mais tarde.

Você precisará agora configurar um sistema de logging. O Hibernate usa logging comuns e lhe oferece a escolha entre o Log4j e o logging do JDK 1.4 . A maioria dos desenvolvedores prefere o Log4j: copie `log4j.properties` da distribuição do Hibernate 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 quiser ter uma saída mais detalhada. Por padrão, apenas as mensagens de inicialização do Hibernate são mostradas no stdout.

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

1.1.7. Carregando e salvando objetos

We are now ready to start doing some real work with Hibernate. Let's start by writing an `EventManager` class with a `main()` method:

```
package org.hibernate.tutorial;

import org.hibernate.Session;

import java.util.*;

import org.hibernate.tutorial.domain.Event;
import org.hibernate.tutorial.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();
    }

}
```

Em `createAndStoreEvent()`, criamos um novo objeto de `Event`, e passamos para o Hibernate. O Hibernate sabe como tomar conta do SQL e executa `INSERT`s no banco de dados.

A org.hibernate.Session is designed to represent a single unit of work (a single atomic piece of work to be performed). For now we will keep things simple and assume a one-to-one granularity between a Hibernate org.hibernate.Session and a database transaction. To shield our code from the actual underlying transaction system we use the Hibernate `org.hibernate.Transaction` API. In this particular case we are using JDBC-based transactional semantics, but it could also run with JTA.

O que a `sessionFactory.getCurrentSession()` faz? Primeiro, você pode chamar quantas vezes e de onde quiser, assim que você receber sua `org.hibernate.SessionFactory`. O método `getCurrentSession()` sempre retorna à unidade de trabalho "atual". Você se lembra que nós mudamos a opção de configuração desse mecanismo para "thread" em nosso `src/main/resources/hibernate.cfg.xml`? Devido a esta configuração, o contexto de uma unidade de trabalho atual estará vinculada à thread Java atual que executa nossa aplicação.

Importante

O Hibernate oferece três métodos da sessão atual. O método "thread" baseado não possui por interesse o uso de produção; ele é basicamente útil para prototyping e tutoriais tais como este. A sessão atual será discutida em mais detalhes mais tarde.

Um org.hibernate.Session começa quando for necessária, quando é feita a primeira chamada à `getCurrentSession()`. É então limitada pelo Hibernate para a thread atual. Quando a transação termina, tanto com commit quanto rollback, o Hibernate também desvincula a `Session` da thread e fecha isso pra você. Se você chamar `getCurrentSession()` novamente, você receberá uma nova `Session` e poderá começar uma nova unidade de trabalho.

Em relação ao escopo da unidade de trabalho, o Hibernate org.hibernate.Session deve ser utilizado para executar uma ou mais operações do banco de dados? O exemplo acima utiliza

uma `Session` para cada operação. Isto é pura coincidência, o exemplo simplesmente não é complexo o bastante para mostrar qualquer outra abordagem. O escopo de um Hibernate org.hibernate.Session é flexível, mas você nunca deve configurar seu aplicativo para utilizar um novo Hibernate org.hibernate.Session para aoperação de banco de dados *every*. Portanto, mesmo que você o veja algumas vezes mais nos seguintes exemplos, considere *session-per-operation* como um anti-modelo. Um aplicativo da web real será demonstrado mais adiante neste tutorial.

See *Capítulo 13, Transações e Concorrência* for more information about transaction handling and demarcation. The previous example also skipped any error handling and rollback.

Para rodar isto, nós faremos uso do Maven exec plugin para chamar nossa classe com a instalação do classpath necessária: `mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="store"`

Nota

Você precisa executar o `mvn compile` primeiramente.

Você deverá ver, após a compilação, a inicialização 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.

Adicionamos uma opção para o método principal com o objetivo de listar os eventos arquivados:

```
        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 `listEvents() method is also added`:

```
    private List listEvents() {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
```

```
        session.beginTransaction();
        List result = session.createQuery("from Event").list();
        session.getTransaction().commit();
        return result;
    }
```

Here, we are using a Hibernate Query Language (HQL) query to load all existing `Event` objects from the database. Hibernate will generate the appropriate SQL, send it to the database and populate `Event` objects with the data. You can create more complex queries with HQL. See *Capítulo 16, HQL: A Linguagem de Consultas do Hibernate* for more information.

Agora podemos chamar nossa nova funcionalidade usando, novamente, o Maven exec plugin: `mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="list"`

1.2. 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 iremos adicionar pessoas à nossa aplicação e armazenar os eventos em que elas participam.

1.2.1. Mapeando a classe Person

O primeira parte da classe `Person` parece-se com isto:

```
package org.hibernate.tutorial.domain;

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'

}
```

Salve isto ao arquivo nomeado `src/main/java/org/hibernate/tutorial/domain/Person.java`

Após isto, crie um novo arquivo de mapeamento como `src/main/resources/org/hibernate/tutorial/domain/Person.hbm.xml`

```
<hibernate-mapping package="org.hibernate.tutorial.domain">
```

```
    <class name="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 à configuração do Hibernate:

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

Crie agora uma associação entre estas duas entidades. As 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.

1.2.2. Uma associação unidirecional baseada em Configuração

Iremos adicionar uma coleção de eventos na classe `Person`. Dessa forma, poderemos navegar pelos eventos de uma pessoa em particular, sem executar uma consulta explicitamente, apenas chamando `Person#getEvents`. As associações de valores múltiplos são representadas no Hibernate por um dos contratos do Java Collection Framework; aqui nós escolhemos um `java.util.Set`, uma vez que a coleção não conterá elementos duplicados e a ordem não é relevante em nossos exemplos:

```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

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

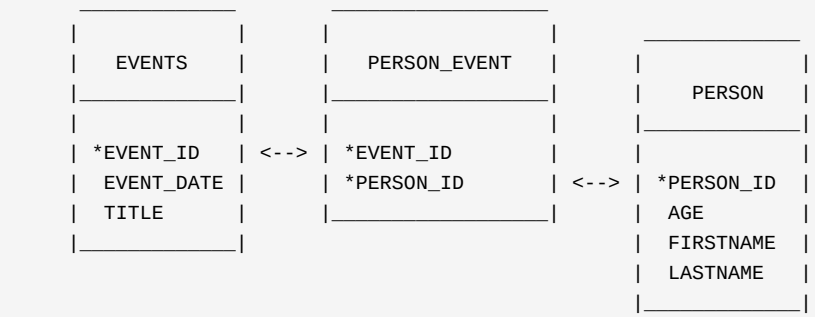
Antes de mapearmos esta associação, pense no outro lado. Claramente, poderíamos apenas fazer isto de forma unidirecional. Ou poderíamos criar outra coleção no `Event`, se quisermos navegar de ambas direções. Isto não é necessário, de uma perspectiva funcional. Você poderá sempre executar uma consulta explícita para recuperar 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" válidos em ambos os lados, nós chamamos isto de uma

associação *muitos-para-muitos*. Daqui pra frente, usaremos o mapeamento muitos-para-muitos do Hibernate:

```
<class name="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="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*, é necessária uma tabela de associação. Cada linha nessa tabela representa um link entre uma pessoa e um evento. O nome da tabela é configurado com a função `table` do elemento `set`. O nome da coluna identificadora na associação, pelo lado da pessoa, é definido com o elemento `key`, o nome da coluna pelo lado dos eventos, é definido com a função `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).

O esquema de mapeamento para o banco de dados está a seguir:



1.2.3. Trabalhando a associação

Vamos reunir algumas pessoas e eventos 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 que necessita ser atualizada. Isso é chamado de *checagem suja automática*, e você também pode usá-la modificando o nome ou a data de qualquer um dos seus objetos. Desde que eles estejam no estado *persistent*, ou seja, limitado por uma `Session` do Hibernate em particular, o Hibernate monitora qualquer alteração e executa o SQL em modo de gravação temporária. O processo de sincronização do estado da memória com o banco de dados, geralmente apenas no final de uma unidade de trabalho, normalmente apenas no final da unidade de trabalho, é chamado de *flushing*. No nosso código, a unidade de trabalho termina com o commit , ou rollback, da transação do banco de dados.

Você pode também querer carregar pessoas e eventos em diferentes unidades de trabalho. Ou você modifica um objeto fora de um `org.hibernate.Session`, quando não se encontra no estado persistente (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 persistente novamente, pode-se dizer que ele liga o objeto a uma nova unidade de trabalho, assim qualquer modificação que você faça neste objeto enquanto estiver no estado desanexado pode ser salvo no banco de dados. Isso inclui qualquer modificação (adição/exclusão) que você faça em uma coleção da entidade deste objeto.

Bem, isso não é de grande utilidade na nossa situação atual, porém, é um importante conceito que você pode criar em seu próprio aplicativo. No momento, complete este exercício adicionando uma ação ao método principal da classe `EventManager` e chame-o pela linha de comando. Se você precisar dos identificadores de uma pessoa ou evento - o método `save()` retornará 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 importante". Alguns você já viu, como um `int` ou uma `String`. Nós chamamos essas classes de *tipos de valores*, 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 tiverem o mesmo objeto firstname. Naturalmente, os tipos de valores não são apenas encontrados dentro da JDK, mas você pode também criar suas classes como, por exemplo, `Address` ou `MonetaryAmount`. De fato, no aplicativo Hibernate todas as classes JDK são consideradas tipos de valores.

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

1.2.4. Coleção de valores

Vamos adicionar uma coleção de endereços de e-mail à entidade `Person`. Isto será representado como um `java.util.Set` das instâncias `java.lang.String`:

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

    public Set getEmailAddresses() {
        return emailAddresses;
    }

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

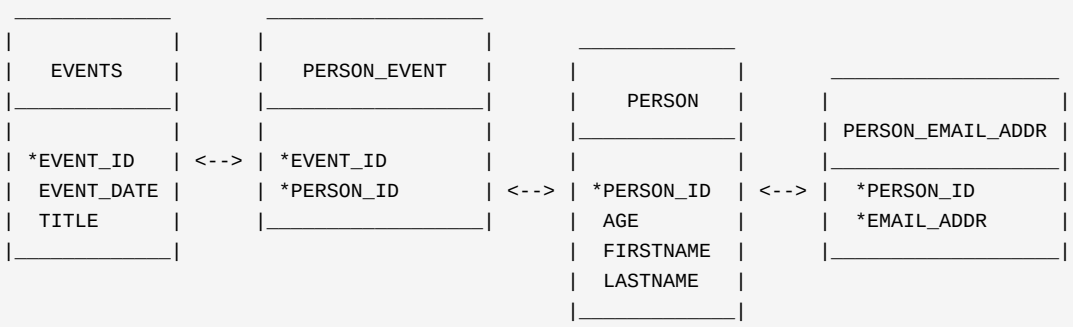
```
    }
```

Segue abaixo o mapeamento deste `Set`:

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

A diferença comparada com o mapeamento anterior se encontra na parte `element`, que informa ao Hibernate que a coleção não contém referências à outra entidade, mas uma coleção de elementos do tipo `String`. O nome da tag em minúsculo indica que se trata de um tipo/conversor de mapeamento do Hibernate. Mais uma vez, a função `table` do elemento `set` determina o nome da tabela para a coleção. O elemento `key` define o nome da coluna de chave estrangeira na tabela de coleção. A função `column` dentro do elemento `element` define o nome da coluna onde os valores da `String` serão armazenados.

Segue abaixo o esquema atualizado:



Você pode observar que a chave primária da tabela da coleção é na verdade uma chave composta, usando as duas 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 à essa coleção, do mesmo modo que fizemos anteriormente ligando pessoas e eventos. É 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);
        // adding to the emailAddress collection might trigger a lazy load of the collection
        aPerson.getEmailAddresses().add(emailAddress);
```

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

Desta vez não utilizamos uma consulta *fetch* (busca) para inicializar a coleção. Monitore o log SQL e tente otimizá-lo com árdua busca.

1.2.5. Associações bidirecionais

Agora iremos mapear uma associação bidirecional. Você fará uma associação entre o trabalho person e event de ambos os lados em Java. O esquema do banco de dados acima não muda, de forma que você continua possuir a multiplicidade muitos-para-muitos.

Nota

Um banco de dados relacional é mais flexível que um linguagem de programação da rede, de maneira que ele não precisa de uma direção de navegação; os dados podem ser visualizados e restaurados de qualquer maneira.

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, esses são mapeamentos `set` normais em ambos documentos de mapeamento. Observe que os nomes das colunas em `key` e `many-to-many` estão trocados em ambos os documentos de mapeamento. A adição mais importante feita está na função `inverse="true"` no elemento `set` da coleção da classe `Event`.

Isso significa que o Hibernate deve pegar o outro lado, a classe `Person`, quando precisar encontrar informação sobre a relação entre as duas entidades. Isso será muito mais fácil de entender quando você analisar como a relação bidirecional entre as entidades é criada.

1.2.6. Trabalhando com links bidirecionais

Primeiro, tenha em mente que o Hibernate não afeta a semântica normal do Java. Como foi que criamos um link entre uma `Person` e um `Event` no exemplo unidirecional? Adicionamos uma instância de `Event`, da coleção de referências de eventos, à uma instância de `Person`. Então, obviamente, se quisermos que este link funcione bidirecionalmente, devemos fazer a mesma coisa para o outro lado, adicionando uma referência de `Person` na coleção de um `Event`. Essa "configuração de link de ambos os lados" é absolutamente necessária e você nunca deve esquecer de fazê-la.

Muitos desenvolvedores programam de maneira defensiva e criam métodos de gerenciamento de um link que ajustam-se corretamente em ambos os lados (como por exemplo, em `Person`):

```
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 coleção estão protegidos. Isso permite que classes e subclasses do mesmo pacote continuem acessando os métodos, mas evita que qualquer outra classe, que não esteja no mesmo pacote, acesse a coleção diretamente. Repita os passos para a coleção do outro lado.

E sobre o mapeamento da função `inverse`? Para você, e para o Java, um link bidirecional é simplesmente uma questão de configurar corretamente as referências de ambos os lados. O Hibernate, entretanto, não possui informação necessária para ajustar corretamente as instruções `INSERT` e `UPDATE` do SQL (para evitar violações de restrição) e precisa de ajuda para manipular as associações bidirecionais de forma apropriada. Ao fazer um lado da associação com a função `inverse`, você instrui o Hibernate para basicamente ignorá-lo, considerando-o uma *cópia* do outro lado. Isso é o necessário para o Hibernate compreender todas as possibilidades quando transformar um modelo de navegação bidirecional em esquema de banco de dados do SQL. As regras que você precisa lembrar são diretas: todas as associações bidirecionais necessitam que

um lado possua a função `inverse`. Em uma associação de um-para-muitos, precisará ser o lado de "muitos", já em uma associação de muitos-para-muitos você poderá selecionar qualquer lado.

1.3. EventManager um aplicativo da web

Um aplicativo de web do Hibernate utiliza uma `Session` e uma `Transaction` quase do mesmo modo que um aplicativo autônomo. Entretanto, alguns modelos 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.

1.3.1. Criando um servlet básico

Nós deveremos criar o nosso servket de processamento básico primeiramente. Uma vez que o servlet manuseia somente requisições `GET` do HTTP, o método que iremos implementar é `doGet()`:

```
package org.hibernate.tutorial.web;

// Imports

public class EventManagerServlet extends HttpServlet {

    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();
            if ( ServletException.class.isInstance( ex ) ) {
                throw ( ServletException ) ex;
            }
            else {
                throw new ServletException( ex );
            }
        }
    }

}
```

Salve esse servlet como `src/main/java/org/hibernate/tutorial/web/EventManagerServlet.java`

O modelo que estamos aplicando neste código é chamado *session-per-request*. Quando uma solicitação 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 e todo acesso a dados deve ocorrer dentro de uma transação, não importando se o dado é de leitura ou escrita. Não se deve utilizar o modo auto-commit em aplicações.

Nunca utilize uma nova `Session` do Hibernate para todas as operações de banco de dados. Utilize uma `Session` do Hibernate que seja de interesse à todas as solicitações. Utilize `getCurrentSession()`, para que seja vinculado automaticamente à thread atual de Java.

Agora, as possíveis ações de uma solicitação serão processadas e uma resposta HTML será renderizada. Já chegaremos nesta parte.

Finalmente, a unidade de trabalho termina quando o processamento e a renderização são completados. Se ocorrer algum erro durante o processamento ou a renderização, uma exceção será lançada e a transação do banco de dados revertida. Isso completa o modelo `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 website do Hibernate e do Wiki para maiores informações sobre esse modelo, chamado *Sessão Aberta na Visualização*. Você precisará disto assim que você considerar renderizar sua visualização no JSP, não apenas num servlet.

1.3.2. Processando e renderizando

Vamos implementar o processamento da solicitação e renderização da página.

```
        // 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 deste código misturado com o Java e HTML, não escalariam em um aplicativo mais complexo, tenha em mente que estamos somente ilustrando os conceitos básicos do Hibernate neste tutorial. O código imprime um cabeçalho e nota de rodapé em HTML. Dentro desta página, são impressos um formulário para entrada de evento em HTML e uma lista de todos os evento no banco de dados. O primeiro método é trivial e somente produz um 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()` utiliza a `Session` do Hibernate, limitado ao thread atual para executar uma consulta:

```
    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>");
            Iterator it = result.iterator();
            while (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 ação `store`, é despachada ao método `createAndStoreEvent()`, que também utiliza 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);
    }
```

O servlet está completo agora. Uma solicitação ao servlet será processada com uma única `Session` e `Transaction`. Quanto antes estiver no aplicativo autônomo, maior a chance do Hibernate vincular automaticamente estes objetos à thread atual de execução. Isto lhe dá a liberdade para inserir seu código e acessar a `SessionFactory` como desejar. Geralmente, usaríamos um diagrama mais sofisticado e moveríamos o código de acesso de dados para os objetos de acesso dos dados (o modelo DAO). Veja o Hibernate Wiki para mais exemplos.

1.3.3. Implementando e testando

Para implementar este aplicativo em testes, nós devemos criar um Arquivo da Web (WAR). Primeiro, nós devemos definir o descritor WAR como `src/main/webapp/WEB-INF/web.xml`

```
<?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.xsd">

    <servlet>
        <servlet-name>Event Manager</servlet-name>
        <servlet-class>org.hibernate.tutorial.web.EventManagerServlet</servlet-class>
    </servlet>

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

Para construir e implementar, chame seu diretório de projeto `ant war` e copie o arquivo `hibernate-tutorial.war` para seu diretório Tomcat `webapp`.

Nota

If you do not have Tomcat installed, download it from *http://tomcat.apache.org/* and follow the installation instructions. Our application requires no changes to the standard Tomcat configuration.

Uma vez implementado e com o Tomcat rodando, acesse o aplicativo em `http://localhost:8080/hibernate-tutorial/eventmanager`. Tenha a certeza de observar o log do Tomcat para ver o Hibernate inicializar quando a primeira solicitação chegar em seu servlet (o

inicializador estático no `HibernateUtil` é chamado) e para obter o resultado detalhado caso exceções aconteçam.

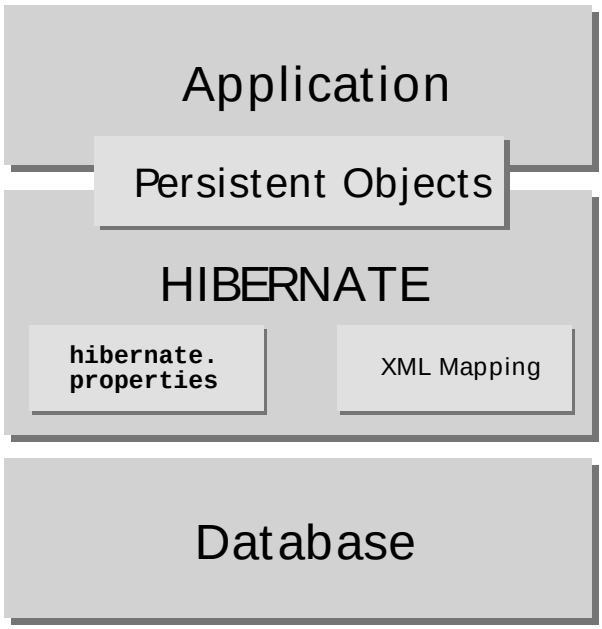
1.4. Sumário

Este tutorial cobriu itens básicos de como escrever um aplicativo Hibernate autônomo simples e um aplicativo da web pequeno. A partir do Hibernate *website* [http://hibernate.org] você poderá encontrar mais tutoriais disponíveis.

Arquitetura

2.1. Visão Geral

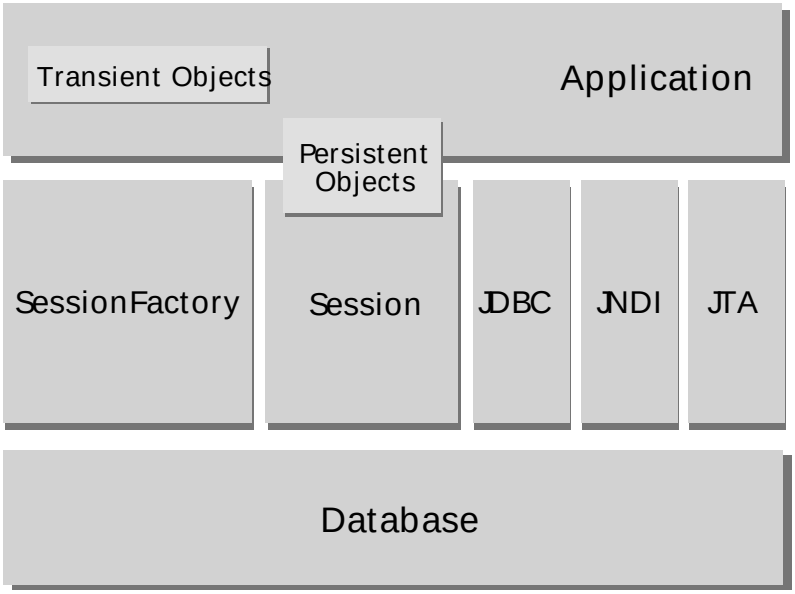
O diagrama abaixo fornece uma visão de altíssimo nível da arquitetura do Hibernate:



Unfortunately we cannot provide a detailed view of all possible runtime architectures. Hibernate is sufficiently flexible to be used in a number of ways in many, many architectures. We will, however, illustrate 2 specifically since they are extremes.

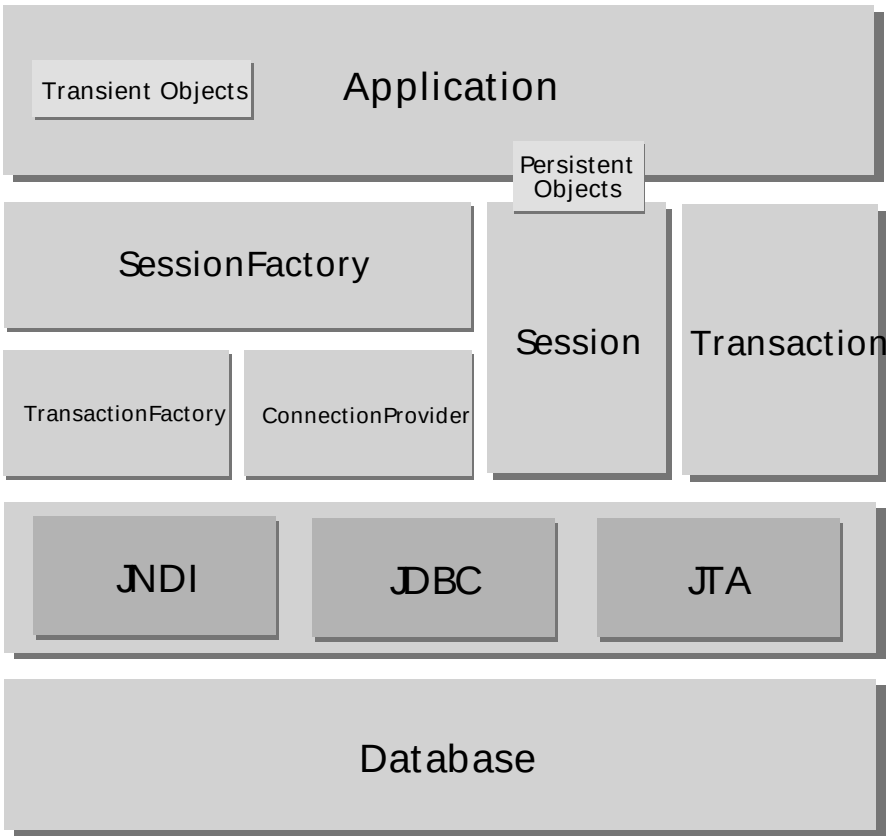
2.1.1. Minimal architecture

The "minimal" architecture has the application manage its own JDBC connections and provide those connections to Hibernate; additionally the application manages transactions for itself. This approach uses a minimal subset of Hibernate APIs.



2.1.2. Comprehensive architecture

A arquitetura "compreensiva" abstrai a aplicação do JDBC/JTA e APIs adjacentes e deixa o Hibernate tomar conta dos detalhes.



2.1.3. Basic APIs

Here are quick discussions about some of the API objects depicted in the preceding diagrams (you will see them again in more detail in later chapters).

SessionFactory (`org.hibernate.SessionFactory`)
: A thread-safe, immutable cache of compiled mappings for a single database. A factory for `org.hibernate.Session` instances. A client of `org.hibernate.connection.ConnectionProvider`. Optionally maintains a `second level cache` of data that is reusable between transactions at a process or cluster level.

Session (`org.hibernate.Session`)
: A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC `java.sql.Connection`. Factory for `org.hibernate.Transaction`. Maintains a `first level cache` of persistent the application's persistent objects and collections; this cache is used when navigating the object graph or looking up objects by identifier.

Objetos persistentes e coleções
: Short-lived, single threaded objects containing persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one `org.hibernate.Session`. Once the `org.hibernate.Session` is closed, they will be detached and free to use in any application layer (for example, directly as data transfer objects to and from presentation). *Capítulo 11, Trabalhando com objetos* discusses transient, persistent and detached object states.

Objetos e coleções desanexados e transientes
: Instances of persistent classes that are not currently associated with a `org.hibernate.Session`. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed `org.hibernate.Session`. *Capítulo 11, Trabalhando com objetos* discusses transient, persistent and detached object states.

Transaction (`org.hibernate.Transaction`)
: (Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC, JTA or CORBA transaction. A `org.hibernate.Session` might span several `org.hibernate.Transactions` in some cases. However, transaction demarcation, either using the underlying API or `org.hibernate.Transaction`, is never optional.

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)
: (Optional) A factory for, and pool of, JDBC connections. It abstracts the application from underlying `javax.sql.DataSource` or `java.sql.DriverManager`. It is not exposed to application, but it can be extended and/or implemented by the developer.

TransactionFactory (`org.hibernate.TransactionFactory`)
: (Optional) A factory for `org.hibernate.Transaction` instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.

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.

2.2. Integração JMX

JMX é o padrão do 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`.

Another feature available as a JMX service is runtime Hibernate statistics. See *Seção 3.4.6, "Estatísticas do Hibernate"* for more information.

2.3. Sessões Contextuais

A maioria das aplicações que usa o Hibernate necessita de algum tipo de sessão "contextual", onde uma sessão dada 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 considerou o uso de transações `JTA`, onde a transação `JTA` definia tanto o escopo quanto o contexto de uma sessão atual. Dada a maturidade de diversas implementações autônomas disponíveis do `JTA TransactionManager`, a maioria (se não todos) dos aplicativos deveria utilizar o gerenciador de transações `JTA` sendo ou não instalados dentro de um recipiente `J2EE`. Baseado neste recurso, você deve sempre utilizar sessões contextuais baseadas em `JTA`.

Entretanto, a partir da versão 3.1, o processo por trás do método `SessionFactory.getCurrentSession()` é agora plugável. 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.

Consulte no Javadocs sobre a interface `org.hibernate.context.CurrentSessionContext` para uma discussão detalhada. Ela define um método único, `currentSession()`, pelo qual a implementação é responsável por rastrear a sessão contextual atual. Fora da caixa, o Hibernate surge com três 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 à abordagem anterior do JTA somente. Consulte em Javadocs para maiores detalhes.
- `org.hibernate.context.ThreadLocalSessionContext` - As sessões correntes são rastreadas por uma thread de execução. Novamente, consulte em Javadocs para maiores detalhes.
- `org.hibernate.context.ManagedSessionContext`. As sessões atuais são rastreadas por uma thread de execução. Entretanto, você é responsável por vincular e desvincular uma instância `Session` com métodos estáticos nesta classe, que nunca abre, libera ou fecha uma `Session`.

The first two implementations provide a "one session - one database transaction" programming model. This is also known and used as *session-per-request*. The beginning and end of a Hibernate session is defined by the duration of a database transaction. If you use programmatic transaction demarcation in plain JSE without JTA, you are advised to use the Hibernate `Transaction` API to hide the underlying transaction system from your code. If you use JTA, you can utilize the JTA interfaces to demarcate transactions. If you execute in an EJB container that supports CMT, transaction boundaries are defined declaratively and you do not need any transaction or session demarcation operations in your code. Refer to *Capítulo 13, Transações e Concorrência* for more information and code examples.

O parâmetro de configuração `hibernate.current_session_context_class` define qual implementação `org.hibernate.context.CurrentSessionContext` deve ser usada. Note que para compatibilidade anterior, se este parâmetro de configuração não for determinado mas um `org.hibernate.transaction.TransactionManagerLookup` for configurado, Hibernate usará o `org.hibernate.context.JTASessionContext`. Tipicamente, o valor deste parâmetro nomearia apenas a classe de implementação para usar; para as três implementações fora da caixa, entretanto, há dois pequenos nomes correspondentes, "jta", "thread", e "managed".

Configuration

Devido ao fato do Hibernate ser projetado para operar em vários ambientes diferentes, há um grande número de parâmetros de configuração. Felizmente, a maioria possui valores padrão consideráveis 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. Configuração programática

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

Você pode obter uma instância `org.hibernate.cfg.Configuration` intanciando-a diretamente e especificando os documentos de mapeamento XML. Se os arquivos de mapeamento estiverem no classpath, use `addResource()`. Por exemplo:

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

Uma alternativa 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);
```

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 que você especifique propriedades de configuração específica. Por exemplo:

```
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");
```

Esta não é a única forma de 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` de arquivo nomeado no diretório raíz do classpath.
3. Determinar as propriedades do `System` usando `java -Dproperty=value`.
4. Incluir elementos `<property>` no `hibernate.cfg.xml` (discutido mais tarde).

Caso você deseje inicializar rapidamente o `hibernate.properties` é a abordagem mais rápida.

O `org.hibernate.cfg.Configuration` é previsto como um objeto de tempo de inicialização, a ser descartado quando um `SessionFactory` for criado.

3.2. Obtendo uma SessionFactory

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

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

O Hibernate permite sua aplicação instanciar mais do que um `org.hibernate.SessionFactory`. Isto será útil se você estiver usando mais do que um banco de dados.

3.3. Conexões JDBC

Normalmente, você deseja que o `org.hibernate.SessionFactory` crie e faça um um pool de conexões JDBC para você. Se você seguir essa abordagem, a abertura de um `org.hibernate.Session` será tão simples quanto:

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

Assim que você fizer algo que requeira o acesso ao banco de dados, uma conexão JDBC será obtida a partir do pool.

Para esse trabalho, precisaremos passar algumas propriedades da conexão JDBC para o Hibernate. Todos os nomes de propriedades Hibernate e semânticas são definidas na classe `org.hibernate.cfg.Environment`. Descreveremos agora as configurações mais importantes para a conexão JDBC.

O Hibernate obterá conexões (e efetuará o 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	*JDBC driver class*
hibernate.connection.url	*JDBC URL*
hibernate.connection.username	*database user*
hibernate.connection.password	*database user password*
hibernate.connection.pool_size	*maximum number of pooled connections*

No entanto, o algoritmo de pool de conexões do próprio Hibernate é um tanto rudimentar. A intenção dele é ajudar a iniciar e *não para ser usado em um sistema de produção* ou até para testar o desempenho. Você deve utilizar um pool de terceiros para conseguir um melhor desempenho e estabilidade. Apenas substitua a propriedade `hibernate.connection.pool_size` pela configuração específica do pool de conexões. Isto irá desligar o pool interno do Hibernate. Por exemplo, você pode gostar de usar C3P0.

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

Este é 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 usar dentro de um servidor de aplicação, você deve configurar o Hibernate para obter conexões de um servidor de aplicação `javax.sql.Datasource` registrado no JNDI. Você precisará determinar pelo menos uma das seguintes propriedades:

Tabela 3.2. Propriedades do Datasource do Hibernate

Nome da Propriedade	Propósito
hibernate.connection.datasource	*datasource JNDI name*
hibernate.jndi.url	*URL do fornecedor JNDI* (opcional)
hibernate.jndi.class	*classe de JNDI* `InitialContextFactory` (opcional)

Nome da Propriedade	Propósito
hibernate.connection.username	*usuário de banco de dados* (opcional)
hibernate.connection.password	*senha de usuário de banco de dados* (opcional)

Eis um exemplo de arquivo `hibernate.properties` para um servidor de aplicação fornecedor de fontes de dados 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 irão automaticamente participar das transações gerenciadas pelo recipiente no servidor de aplicação.

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

Você pode definir sua própria estratégia de plugin para obter conexões JDBC implementando a interface `org.hibernate.connection.ConnectionProvider` e especificando sua implementação customizada através da propriedade `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 têm valores padrão lógicos.

Atenção

Algumas destas propriedades são somente em nível de sistema.. As propriedades em nível de sistema podem ser determinadas somente via `java -Dproperty=value` ou `hibernate.properties`. Elas *não podem* ser configuradas por outras técnicas descritas acima.

Tabela 3.3. Propriedades de Configuração do Hibernate

Nome da Propriedade	Propósito
hibernate.dialect	O nome da classe de um Hibernate `org.hibernate.dialect.Dialect` que permite o Hibernate gerar SQL otimizado

Nome da Propriedade	Propósito
	para um banco de dados relacional em particular. **e.g.** `full.classname.of.Dialect` Na maioria dos casos, o Hibernate irá atualmente estar apto a escolher a implementação `org.hibernate.dialect.Dialect` correta baseada no `JDBC metadata` retornado pelo JDBC driver.
hibernate.show_sql	Escreve todas as instruções SQL no console. Esta é uma alternativa para configurar a categoria de log `org.hibernate.SQL` to `debug`. **e.g.** `true` | `false`
hibernate.format_sql	Imprime o SQL formatado no log e recipiente. **e.g.** `true` | `false`
hibernate.default_schema	Qualifica no SQL gerado, os nome das tabelas desqualificadas com o esquema/espaço da tabela dado. **e.g.** `SCHEMA_NAME`
hibernate.default_catalog	Qualifica no SQL gerado, os nome das tabelas desqualificadas com catálogo dado. **e.g.** `CATALOG_NAME`
hibernate.session_factory_name	O `org.hibernate.SessionFactory` irá automaticamente se ligar a este nome no JNDI depois de ter sido criado. **e.g.** `jndi/composite/name`
hibernate.max_fetch_depth	Estabelece a "profundidade" máxima para árvore de busca de união externa para associações finais únicas (um para um, muitos para um). Um `0` desativa por padrão a busca de união externa. **eg.** valores recomendados entre`0` e `3`
hibernate.default_batch_fetch_size	Determina um tamanho padrão para busca de associações em lotes do Hibernate.

Nome da Propriedade	Propósito
	eg. valores recomendados `4`, `8`, `16`
hibernate.default_entity_mode	Determina um modo padrão para representação de entidade para todas as sessões abertas desta `SessionFactory` `dynamic-map`, `dom4j`, `pojo`
hibernate.order_updates	Força o Hibernate a ordenar os updates SQL pelo valor da chave primária dos ítens a serem atualizados. Isto resultará em menos deadlocks nas transações em sistemas altamente concorrente. **e.g.** `true` | `false`
hibernate.generate_statistics	Se habilitado, o Hibernate coletará estatísticas úteis para o ajuste do desempenho. **e.g.** `true` | `false`
hibernate.use_identifier_rollback	Se habilitado, propriedades identificadoras geradas serão zeradas para os valores padrão quando os objetos forem apagados. **e.g.** `true` | `false`
hibernate.use_sql_comments	Se ligado, o Hibernate irá gerar comentários dentro do SQL, para facilitar a depuração, o valor padrão é `false` **e.g.** `true` | `false`
hibernate.id.new_generator_mappings	Setting is relevant when using `@GeneratedValue`. It indicates whether or not the new `IdentifierGenerator` implementations are used for `javax.persistence.GenerationType.AUTO`, `javax.persistence.GenerationType.TABLE` and `javax.persistence.GenerationType.SEQUENCE`. Default to `false` to keep backward compatibility. **e.g.** `true` | `false`

Nota

We recommend all new projects which make use of to use `@GeneratedValue` to also set `hibernate.id.new_generator_mappings=true` as the new generators are more efficient and closer to the JPA 2 specification semantic. However they are not backward compatible with existing databases (if a sequence or a table is used for id generation).

Tabela 3.4. JDBC Hibernate e Propriedades de Conexão

Nome da Propriedade	Propósito
hibernate.jdbc.fetch_size	Um valor maior que zero determina o tamanho da buscado JDBC (chamadas `Statement.setFetchSize()`).
hibernate.jdbc.batch_size	Um valor maior que zero habilita o uso das atualizações em lotes JDBC2 pelo Hibernate. **ex.** valores recomentados entre `5` e `30`
hibernate.jdbc.batch_versioned_data	Set this property to `true` if your JDBC driver returns correct row counts from `executeBatch()`. It is usually safe to turn this option on. Hibernate will then use batched DML for automatically versioned data. Defaults to `false`. **e.g.** `true` | `false`
hibernate.jdbc.factory_class	Escolher um `org.hibernate.jdbc.Batcher`. Muitas aplicações não irão precisar desta propriedade de configuração. **exemplo** `classname.of.BatcherFactory`
hibernate.jdbc.use_scrollable_resultset	Habilita o uso dos resultados de ajustes roláveis do JDBC2 pelo Hibernate. Essa propriedade somente é necessária quando se usa Conexões JDBC providas pelo usuário. Do contrário, o Hibernate os os metadados da conexão. **e.g.** `true` | `false`
hibernate.jdbc.use_streams_for_binary	Utilize fluxos para escrever/ler tipos `binary` ou tipos `serializable` para/do JDBC. **system-level property**

Nome da Propriedade	Propósito
	e.g. `true` | `false`
hibernate.jdbc.use_get_generated_keys	Possibilita o uso do `PreparedStatement.getGeneratedKeys()` JDBC3 para recuperar chaves geradas de forma nativa depois da inserção. Requer driver JDBC3+ e JRE1.4+, ajuste para falso se seu driver tiver problemas com gerador de indentificadores Hibernate. Por padrão, tente determinar o driver capaz de usar metadados da conexão. **exemplo** `true|false`
hibernate.connection.provider_class	O nome da classe de um `org.hibernate.connection.ConnectionProvider`, do qual proverá conexões JDBC para o Hibernate. **exemplo**`classname.of.ConnectionProvider`
hibernate.connection.isolation	Determina o nível de isolamento de uma transação JDBC. Verifique `java.sql.Connection` para valores significativos mas note que a maior parte dos bancos de dados não suportam todos os isolamentos que não são padrões. **exemplo** `1, 2, 4, 8`
hibernate.connection.autocommit	Habilita o auto-commit para conexões no pool JDBC (não recomendado). **e.g.** `true` | `false`
hibernate.connection.release_mode	Especifica quando o Hibernate deve liberar conexões JDBC. Por padrão, uma conexão JDBC é retida até a sessão estar explicitamente fechada ou desconectada. Para uma fonte de dados JTA do servidor de aplicação, você deve usar `after_statement` para forçar a liberação da conexões depois de todas as chamadas JDBC. Para uma conexão não-JTA, freqüentemente faz sentido liberar a conexão ao fim de cada transação, usando `after_transaction`. O `auto` escolherá `after_statement` para as estratégias de transaçãoes JTA e CMT e

Nome da Propriedade	Propósito
	`after_transaction` para as estratégias de transação JDBC. **exemplo** `auto` (padrão) | `on_close` | `after_transaction` | `after_statement` This setting only affects `Sessions` returned from `SessionFactory.openSession`. For `Sessions` obtained through `SessionFactory.getCurrentSession`, the `CurrentSessionContext` implementation configured for use controls the connection release mode for those `Sessions`. See *Seção 2.3, "Sessões Contextuais"*
hibernate.connection.*<propertyName>*	Passar a propriedade JDBC *<propertyName>* para `DriverManager.getConnection()`.
hibernate.jndi.*<propertyName>*	Passar a propriedade *<propertyName>* para o JNDI `InitialContextFactory`.

Tabela 3.5. Propriedades de Cachê do Hibernate

Nome da Propriedade	Propósito
`hibernate.cache.provider_class`	O nome da classe de um `CacheProvider` personalizado. **exemplo** `classname.of.CacheProvider`
`hibernate.cache.use_minimal_puts`	Otimizar operação de cachê de segundo nível para minimizar escritas, ao custo de leituras mais freqüentes. Esta configuração é mais útil para cachês em cluster e, no Hibernate3, é habilitado por padrão para implementações de cache em cluster. **exemplo** `true|false`
`hibernate.cache.use_query_cache`	Habilita a cache de consultas. Mesmo assim, consultas individuais ainda têm que ser habilitadas para o cache. **exemplo** `true|false`
`hibernate.cache.use_second_level_cache`	Pode ser utilizado para desabilitar completamente o cache de segundo nível, o qual é habilitado por padrão para as classes que especificam um mapeamento `<cache>`.

Nome da Propriedade	Propósito
	exemplo `true|false`
`hibernate.cache.query_cache_factory`	O nome de classe de uma interface personalizada `QueryCache`, padroniza para o `StandardQueryCache` criado automaticamente. **exemplo** `classname.of.QueryCache`
`hibernate.cache.region_prefix`	Um prefixo para usar em nomes regionais de cachê de segundo nível **exemplo** `prefix`
`hibernate.cache.use_structured_entries`	Força o Hibernate a armazenar dados no cachê de segundo nível em um formato mais humanamente amigável. **exemplo** `true|false`
`hibernate.cache.default_cache_concurrency_strategy`	Setting used to give the name of the default `org.hibernate.annotations.CacheConcurrencyStrategy` to use when either `@Cacheable` or `@Cache` is used. `@Cache(strategy="..")` is used to override this default.

Tabela 3.6. Propriedades de Transação do Hibernate

Nome da Propriedade	Propósito
`hibernate.transaction.factory_class`	O nome da classe de uma `TransactionFactory` para usar com API do Hibernate `Transaction` (por padrão`JDBCTransactionFactory`). **exemplo** `classname.of.TransactionFactory`
`jta.UserTransaction`	Um nome JNDI usado pelo `JTATransactionFactory`para obter uma `UserTransaction` JTA a partir do servidor de aplicação. **e.g.** `jndi/composite/name`
`hibernate.transaction.manager_lookup_class`	O nome da classe de um `TransactionManagerLookup`. Ele é requerido quando o caching a nível JVM estiver habilitado ou quando estivermos usando um gerador hilo em um ambiente JTA.

Nome da Propriedade	Propósito
	exemplo `classname.of.TransactionManagerLookup`
`hibernate.transaction.flush_before_completion`	If enabled, the session will be automatically flushed during the before completion phase of the transaction. Built-in and automatic session context management is preferred, see *Seção 2.3, "Sessões Contextuais"*. **e.g.** `true` | `false`
`hibernate.transaction.auto_close_session`	If enabled, the session will be automatically closed during the after completion phase of the transaction. Built-in and automatic session context management is preferred, see *Seção 2.3, "Sessões Contextuais"*. **e.g.** `true` | `false`

Tabela 3.7. Propriedades Variadas

Nome da Propriedade	Propósito
`hibernate.current_session_context_class`	Supply a custom strategy for the scoping of the "current" `Session`. See *Seção 2.3, "Sessões Contextuais"* for more information about the built-in strategies. **exemplo** `jta` | `thread` | `managed` | `custom.Class`
`hibernate.query.factory_class`	Escolha a implementação de análise HQL. **exemplo** `org.hibernate.hql.ast.ASTQueryTranslatorFactory` ou `org.hibernate.hql.classic.ClassicQueryTranslatorFactory`
`hibernate.query.substitutions`	Mapeamento a partir de símbolos em consultas do Hibernate para para símbolos SQL (símbolos devem ser por exemplo, funções ou nome literais). **exemplo** `hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC`
`hibernate.hbm2ddl.auto`	Automaticamente valida ou exporta DDL esquema para o banco de dados quando o `SessionFactory` é criado. Com `create-drop`, o esquema do banco de dados

Nome da Propriedade	Propósito
	será excluido quando a `SessionFactory` for fechada explicitamente. **exemplo** `validate` | `update` | `create` | `create-drop`
`hibernate.hbm2ddl.import_file`	Comma-separated names of the optional files containing SQL DML statements executed during the `SessionFactory` creation. This is useful for testing or demoing: by adding INSERT statements for example you can populate your database with a minimal set of data when it is deployed. File order matters, the statements of a give file are executed before the statements of the following files. These statements are only executed if the schema is created ie if `hibernate.hbm2ddl.auto` is set to `create` or `create-drop`. **e.g.** `/humans.sql,/dogs.sql`
`hibernate.bytecode.use_reflection_optimizer`	Enables the use of bytecode manipulation instead of runtime reflection. This is a System-level property and cannot be set in `hibernate.cfg.xml`. Reflection can sometimes be useful when troubleshooting. Hibernate always requires either CGLIB or javassist even if you turn off the optimizer. **e.g.** `true` | `false`
`hibernate.bytecode.provider`	Both javassist or cglib can be used as byte manipulation engines; the default is `javassist`. **e.g.** `javassist` | `cglib`

3.4.1. Dialetos SQL

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

Tabela 3.8. Dialetos SQL do Hibernate (`hibernate.dialect`)

RDBMS	Dialeto
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
Meu SQL	`org.hibernate.dialect.MySQLDialect`
MeuSQL com InnoDB	`org.hibernate.dialect.MySQLInnoDBDialect`
Meu SQL com MeuISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle (qualquer versão)	`org.hibernate.dialect.OracleDialect`
Oracle 9i	`org.hibernate.dialect.Oracle9iDialect`
Oracle 10g	`org.hibernate.dialect.Oracle10gDialect`
Sybase	`org.hibernate.dialect.SybaseDialect`
Sybase Qualquer lugar	`org.hibernate.dialect.SybaseAnywhereDialect`
Servidor Microsoft SQL	`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`
Progresso	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Base Ponto	`org.hibernate.dialect.PointbaseDialect`
Base Frontal	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

3.4.2. Busca por união externa (Outer Join Fetching)

Se seu banco de dados suporta união externa no estilo ANSI, Oracle ou Sybase, a *outer join fetching* freqüentemente aumentará o desempenho limitando o número de chamadas (round trips) para e a partir do banco de dados. No entanto, isto ao custo de possivelmente mais trabalho desempenhado pelo próprio banco de dados. A busca por união externa (outer join fetching) permite um gráfico completo de objetos conectados por associações muitos-para-um, um-para-muitos, muitos-para-muitos e um-para-um para serem recuperadas em uma simples instrução SQL `SELECT`.

A busca por união externa pode ser desabilitada *globalmente* configurando a propriedade `hibernate.max_fetch_depth` para `0`. Um valor `1` ou maior habilita a busca por união externa para associações um-para-um e muitos-para-um, cujos quais têm sido mapeados com `fetch="join"`.

See *Seção 21.1, "Estratégias de Busca "* for more information.

3.4.3. Fluxos Binários (Binary Streams)

O Oracle limita o tamanho de matrizes de `byte` que podem ser passadas para/do 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 em nível de sistema.*

3.4.4. Cachê de segundo nível e consulta

The properties prefixed by `hibernate.cache` allow you to use a process or cluster scoped second-level cache system with Hibernate. See the *Seção 21.2, "O Cachê de Segundo Nível"* for more information.

3.4.5. Substituição na Linguagem de Consulta

Você pode definir novos símbolos de consulta Hibernate usando `hibernate.query.substitutions`. Por exemplo:

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

Isto faria com que os símbolos `true` e `false` passasem a ser traduzidos para literais inteiros no SQL gerado.

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

Isto permitirá que você renomeie 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 ao ajustar um 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

O Hibernate utiliza o *Simple Logging Facade for Java* [http://www.slf4j.org/] (SLF4J) com o objetivo de registrar os diversos eventos de sistema. O SLF4J pode direcionar a sua saída de

logging a diversos frameworks de logging (NOP, Simple, log4j version 1.2, JDK 1.4 logging, JCL ou logback) dependendo de sua escolha de vinculação. Com o objetivo de determinar o seu logging, você precisará do `slf4j-api.jar` em seu classpatch juntamente com o arquivo jar para a sua vinculação preferida - `slf4j-log4j12.jar` no caso do Log4J. Consulte o SLF4J *documentation* [http://www.slf4j.org/manual.html] para maiores detalhes. Para usar o Log4j você precisará também colocar um arquivo `log4j.properties` em seu classpath. Um exemplo do arquivo de propriedades está distribuído com o Hibernate no diretório `src/`.

Nós recomendamos que você se familiarize-se com mensagens de log do Hibernate. Tem sido um árduo trabalho fazer o log Hibernate tão detalhado quanto possível, sem fazê-lo ilegível. É um dispositivo de controle de erros essencial. 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 à sessão no momento da liberação (flush).
`org.hibernate.cache`	Registra todas as atividades de cachê de segundo nível
`org.hibernate.transaction`	Registra atividades relacionada à 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. Apesar de ter muita informação, é muito útil para o problema de inicialização.

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

3.6. Implementando um `NamingStrategy`

A interface `org.hibernate.cfg.NamingStrategy` permite que você especifique um "padrão de nomeação" para objetos do banco de dados e elementos de esquema.

Você deve criar regras para a geração automaticamente de identificadores do banco de dados a partir de identificadores Java ou para processar colunas "lógicas" e nomes de tabelas dado o arquivo de mapeamento para nomes "físicos" de tabelas e colunas. Este recurso ajuda a reduzir a

verbosidade do documento de mapeamento, eliminando interferências repetitivas (`TBL_` prefixos, por exemplo). A estratégia padrão usada pelo Hibernate é bastante 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 inicial 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 estiverem presentes, para substituir propriedades.

O arquivo XML de configuração deve ser encontrado na raíz do seu `CLASSPATH`. Veja um exemplo:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://www.hibernate.org/dtd/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 equivalentes, exceto os acima mencionados de usar a sintaxe de XML.

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

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

Você poderá escolher um arquivo de configuração XML diferente, utilizando:

```
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 a infraestrutura de 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 programaticamente (BMT) ou você poderia querer usar a API opcional do Hibernate `Transaction` para esta manter seu código portável.

- *Vinculação (binding) automática à JNDI*: O Hibernate pode associar sua `SessionFactory` a JNDI depois de iniciado.

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

- *JMX deployment:* Se você usa um JMX servidor de aplicações capaz (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 também cuidará das dependências de serviços (DataSources, têm que estar disponíveis antes do Hibernate iniciar, etc.).

Dependendo do 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 "retenção de conexão".

3.8.1. Configuração de estratégia de transação

A API Hibernate `Session` é independente de qualquer sistema de demarcação 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 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 uma classe construtora para instâncias `Transaction` ajustando a propriedade de configuração do `hibernate.transaction.factory_class`.

Existem três escolhas, ou internas, padrões:

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

`org.hibernate.transaction.JTATransactionFactory`
: delega para uma transação à um container gerenciado se uma transação existente estiver de acordo neste contexto (ex: método bean de sessão EJB). No entanto, uma nova transação será iniciada e serão usadas transações controladas por um bean.

`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

Factory de Transação	Servidor de Aplicação
`org.hibernate.transaction.JBossTransactionManagerLookup`	JBoss AS
`org.hibernate.transaction.WeblogicTransactionManagerLookup`	Weblogic
`org.hibernate.transaction.WebSphereTransactionManagerLookup`	WebSphere

Factory de Transação	Servidor de Aplicação
`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
`org.hibernate.transaction.JRun4TransactionManagerLookup`	JRun4
`org.hibernate.transaction.BESTransactionManagerLookup`	Borland ES
`org.hibernate.transaction.JBossTSStandaloneTransactionManagerLookup`	JBoss TS used standalone (ie. outside JBoss AS and a JNDI environment generally). Known to work for `org.jboss.jbossts:jbossjta:4.11.0.Final`

3.8.2. `SessionFactory` vinculada à JNDI

Uma `SessionFactory` de Hibernate vinculada à JNDI pode simplificar a localização da fábrica e a criação de novas `Session`s. Observe que isto não está relacionado a um `Datasource` ligado a JNDI, simplesmente ambos usam o mesmo registro.

Se você desejar ter uma `SessionFactory` limitada a um nome de espaço de JNDI, especifique um nome (ex.: `java:hibernate/SessionFactory`) usando a propriedade `hibernate.session_factory_name`. Se esta propriedade for omitida, a `SessionFactory` não será limitada ao JNDI. Isto é muito útil em ambientes com uma implementação padrão JNDI de somente leitura (ex.: Tomcat).

Ao vincular a `SessionFactory` ao JNDI, o Hibernate irá utilizar os valores de `hibernate.jndi.url`, `hibernate.jndi.class` para instanciar um contexto inicial. Se eles não forem especificados, será usado o padrão `InitialContext`.

O Hibernate colocará automaticamente a `SessionFactory` no JNDI depois que você chamar a `cfg.buildSessionFactory()`. Isto significa que você terá esta chamada em pelo menos algum código de inicialização (ou classe de utilidade) em seu aplicativo, a não ser que você use a implementação JMX com o `HibernateService` (discutido mais tarde).

Se você usar um JNDI `SessionFactory`, o EJB ou qualquer outra classe obterá a `SessionFactory` utilizando um localizador JNDI.

It is recommended that you bind the `SessionFactory` to JNDI in a managed environment and use a `static` singleton otherwise. To shield your application code from these details, we also recommend to hide the actual lookup code for a `SessionFactory` in a helper class, such as `HibernateUtil.getSessionFactory()`. Note that such a class is also a convenient way to startup Hibernate—see chapter 1.

3.8.3. Gerenciamento de contexto de Sessão atual com JTA

The easiest way to handle `Sessions` and transactions is Hibernate's automatic "current" `Session` management. For a discussion of contextual sessions see *Seção 2.3, "Sessões Contextuais"*. Using the `"jta"` session context, if there is no Hibernate `Session` associated with the current JTA transaction, one will be started and associated with that JTA transaction the first time you call `sessionFactory.getCurrentSession()`. The `Sessions` retrieved via `getCurrentSession()` in the `"jta"` context are set to automatically flush before the transaction completes, close after the transaction completes, and aggressively release JDBC connections after each statement. This allows the `Sessions` to be managed by the life cycle of the JTA transaction to which it is associated, keeping user code clean of such management concerns. Your code can either use JTA programmatically through `UserTransaction`, or (recommended for portable code) use the Hibernate `Transaction` API to set transaction boundaries. If you run in an EJB container, declarative transaction demarcation with CMT is preferred.

3.8.4. implementação JMX

A linha `cfg.buildSessionFactory()` ainda precisa ser executada em algum local para conseguir uma `SessionFactory` em JNDI. Você pode escolher fazer isto em um bloqueio de inicializador `static`, como aquele em `HibernateUtil`, ou implementar o Hibernate como *serviço gerenciado*.

O Hibernate é distribuído com o `org.hibernate.jmx.HibernateService` para implementação em um servidor de aplicativo com capacidades JMX, tal como o JBoss AS. A implementação atual e configuração é comercial. Segue aqui um exemplo do `jboss-service.xml` para o 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 implementado em um diretório chamado `META-INF` e envolto em um arquivo JAR com a extensão `.sar` (arquivo de serviço). Você também pode precisar envolver o Hibernate, suas bibliotecas de terceiros solicitadas, suas classes persistentes compiladas, assim como seus arquivos de mapeamento no mesmo arquivo. Seus beans de empresa (geralmente beans de sessão) podem ser mantidos em seus próprios arquivos JAR, mas você poderá incluir estes arquivos EJB JAR no arquivo de serviço principal para conseguir uma única unidade de (hot)-deployable. Consulte a documentação do JBoss AS para maiores informações sobre o serviço JMX e implementação EJB.

Classes Persistentes

Persistent classes are classes in an application that implement the entities of the business problem (e.g. Customer and Order in an E-commerce application). The term "persistent" here means that the classes are able to be persisted, not that they are in the persistent state (see *Seção 11.1, "Estado dos objetos no Hibernate"* for discussion).

Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model. However, none of these rules are hard requirements. Indeed, Hibernate assumes very little about the nature of your persistent objects. You can express a domain model in other ways (using trees of `java.util.Map` instances, for example).

4.1. Um exemplo simples de POJO

Exemplo 4.1. Simple POJO representing a cat

```
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);
    }
}
```

As quatro regras principais das classes persistentes são descritas em maiores detalhes nas seguintes seções.

4.1.1. Implemente um construtor de não argumento

`Cat` has a no-argument constructor. All persistent classes must have a default constructor (which can be non-public) so that Hibernate can instantiate them using `java.lang.reflect.Constructor.newInstance()`. It is recommended that this constructor be defined with at least *package* visibility in order for runtime proxy generation to work properly.

4.1.2. Provide an identifier property

Nota

Historically this was considered option. While still not (yet) enforced, this should be considered a deprecated feature as it will be completely required to provide a identifier property in an upcoming release.

`Cat` has a property named `id`. This property maps to the primary key column(s) of the underlying database table. The type of the identifier property can be any "basic" type (see *???*). See *Seção 9.4, "Componentes como identificadores compostos"* for information on mapping composite (multi-column) identifiers.

Nota

Identifiers do not necessarily need to identify column(s) in the database physically defined as a primary key. They should just identify columns that can be used to uniquely identify rows in the underlying table.

Recomendamos que você declare propriedades de identificador nomeados de forma consistente nas classes persistentes e que você use um tipo anulável (ou seja, não primitivo).

4.1.3. Prefer non-final classes (semi-optional)

A central feature of Hibernate, *proxies* (lazy loading), depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods. You can persist `final` classes that do not implement an interface with Hibernate; you will not, however, be able to use proxies for lazy association fetching which will ultimately limit your options for performance tuning. To persist a `final` class which does not implement a "full" interface you must disable proxy generation. See *Exemplo 4.2, "Disabling proxies in hbm.xml"* and *Exemplo 4.3, "Disabling proxies in annotations"*.

Exemplo 4.2. Disabling proxies in `hbm.xml`

```
<class name="Cat" lazy="false"...>...</class>
```

Exemplo 4.3. Disabling proxies in annotations

```
@Entity @Proxy(lazy=false) public class Cat { ... }
```

If the `final` class does implement a proper interface, you could alternatively tell Hibernate to use the interface instead when generating the proxies. See *Exemplo 4.4, "Proxying an interface in hbm.xml"* and *Exemplo 4.5, "Proxying an interface in annotations"*.

Exemplo 4.4. Proxying an interface in `hbm.xml`

```
<class name="Cat" proxy="ICat"...>...</class>
```

Exemplo 4.5. Proxying an interface in annotations

```
@Entity @Proxy(proxyClass=ICat.class) public class Cat implements ICat { ... }
```

You should also avoid declaring `public final` methods as this will again limit the ability to generate *proxies* from this class. If you want to use a class with `public final` methods, you must explicitly disable proxying. Again, see *Exemplo 4.2, "Disabling proxies in hbm.xml"* and *Exemplo 4.3, "Disabling proxies in annotations"*.

4.1.4. Declare acessores e mutadores para campos persistentes (opcional)

`Cat` declares accessor methods for all its persistent fields. Many other ORM tools directly persist instance variables. It is better to provide an indirection between the relational schema and internal data structures of the class. By default, Hibernate persists JavaBeans style properties and recognizes method names of the form `getFoo`, `isFoo` and `setFoo`. If required, you can switch to direct field access for particular properties.

Properties need *not* be declared public. Hibernate can persist a property declared with `package`, `protected` or `private` visibility as well.

4.2. Implementando herança

Uma subclasse também deve observar as primeiras e segundas regras. Ela herda sua propriedade de identificador a partir das superclasses, `Cat`. Por exemplo:

```
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 substituir os métodos `equals()` e `hashCode()` se você:

- pretender inserir instâncias de classes persistentes em um `Set` (a forma mais recomendada é representar associações de muitos valores), *e*
- pretender usar reconexão de instâncias desanexadas

O Hibernate garante a equivalência de identidades persistentes (linha de base de dados) e identidade Java somente dentro de um certo escopo de sessão. Dessa forma, assim que misturarmos instâncias recuperadas em sessões diferentes, devemos implementar `equals()` e `hashCode()` se quisermos ter semânticas significativas para os `Sets`.

A forma mais óbvia é implementar `equals()`/`hashCode()` comparando o valor do identificador de ambos objetos. Caso o valor seja o mesmo, ambos devem ter a mesma linha de base de dados, assim eles serão iguais (se ambos forem adicionados a um `Set`, nós só teremos um elemento no `Set`). Infelizmente, não podemos usar esta abordagem com os identificadores gerados. O Hibernate atribuirá somente os valores de identificadores aos objetos que forem persistentes, uma instância recentemente criada não terá nenhum valor de identificador. Além disso, se uma instância não for salva e estiver em um `Set`, salvá-la atribuirá um valor de identificador ao objeto. Se `equals()` e `hashCode()` fossem baseados em um valor identificador, o código hash teria mudado, quebrando o contrato do `Set`. Consulte o website do Hibernate para acessar uma discussão completa sobre este problema. Note que esta não é uma edição do Hibernate, e sim semânticas naturais do Java de igualdade e identidade.

Recomendamos implementar `equals()` e `hashCode()` usando *Business key equality*. A chave de negócios significa que o método `equals()` compara somente a propriedade que formar uma chave de negócios, uma chave que identificaria nossa instância na realidade (uma chave de candidato *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;
    }

}
```

A business key does not have to be as solid as a database primary key candidate (see *Seção 13.1.3, "Considerando a identidade do objeto"*). Immutable or unique properties are usually good candidates for a business key.

4.4. Modelos dinâmicos

Nota

The following features are currently considered experimental and may change in the near future.

Entidades persistentes não precisam ser representadas como classes POJO ou como objetos JavaBeans em tempo de espera. O Hibernate também suporta modelos dinâmicos (usando `Maps` de `Maps` em tempo de execução) e a representação de entidades como árvores DOM4J. Com esta abordagem, você não escreve classes persistes, somente arquivos de mapeamentos.

By default, Hibernate works in normal POJO mode. You can set a default entity representation mode for a particular `SessionFactory` using the `default_entity_mode` configuration option (see *Tabela 3.3, "Propriedades de Configuração do Hibernate"*).

Os seguintes exemplos demonstram a representação usando `Maps`. Primeiro, no arquivo de mapeamento, um `entity-name` precisa ser declarado ao invés 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>
```

Note que embora as associações sejam declaradas utilizando nomes de classe, o tipo alvo de uma associação pode também ser uma entidade dinâmica, ao invés de um POJO.

Após ajustar o modo de entidade padrão para `dynamic-map` para a `SessionFactory`, você poderá trabalhar com `Maps` de `Maps` no período de execução:

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

// 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 de um mapeamento dinâmico são o tempo de retorno rápido para realizar o protótipo sem a necessidade de implementar uma classe de entidade. No entanto, você perde o tipo de tempo de compilação, verificando e muito provavelmente terá que lidar com muitas exceções de tempo de espera. Graças ao mapeamento do Hibernate, o esquema do banco de dados pode ser facilmente normalizado e seguro, permitindo adicionar uma implementação modelo de domínio apropriado na camada do topo num futuro próximo.

Modos de representação de entidade podem ser também ajustados para base por `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
```

Por favor, note que a chamada para a `getSession()` usando um `EntityMode` está na API de `Session` e não na `SessionFactory`. Dessa forma, a nova `Session` compartilha a conexão, transação e outra informação de contexto JDBC adjacente. Isto significa que você não precisará chamar `flush()` e `close()` na `Session` secundária, e também deixar a transação e o manuseio da conexão para a unidade primária do trabalho.

More information about the XML representation capabilities can be found in *Capítulo 20, Mapeamento XML*.

4.5. Tuplizadores

`org.hibernate.tuple.Tuplizer` and its sub-interfaces are responsible for managing a particular representation of a piece of data given that representation's `org.hibernate.EntityMode`. If a given piece of data is thought of as a data structure, then a tuplizer is the thing that knows how to create such a data structure, how to extract values from such a data structure and how to inject values into such a data structure. For example, for the POJO entity mode, the corresponding tuplizer knows how create the POJO through its constructor. It also knows how to access the POJO properties using the defined property accessors.

There are two (high-level) types of Tuplizers:

- `org.hibernate.tuple.entity.EntityTuplizer` which is responsible for managing the above mentioned contracts in regards to entities
- `org.hibernate.tuple.component.ComponentTuplizer` which does the same for components

Users can also plug in their own tuplizers. Perhaps you require that `java.util.Map` implementation other than `java.util.HashMap` be used while in the dynamic-map entity-mode. Or perhaps you need to define a different proxy generation strategy than the one used by default. Both would be achieved by defining a custom tuplizer implementation. Tuplizer definitions are attached to the entity or component mapping they are meant to manage. Going back to the example of our `Customer` entity, *Exemplo 4.6, “Specify custom tuplizers in annotations”* shows how to specify a custom `org.hibernate.tuple.entity.EntityTuplizer` using annotations while *Exemplo 4.7, “Specify custom tuplizers in hbm.xml”* shows how to do the same in `hbm.xml`

Exemplo 4.6. Specify custom tuplizers in annotations

```
@Entity
@Tuplizer(impl = DynamicEntityTuplizer.class)
public interface Cuisine {
    @Id
    @GeneratedValue
    public Long getId();
    public void setId(Long id);

    public String getName();
    public void setName(String name);

    @Tuplizer(impl = DynamicComponentTuplizer.class)
    public Country getCountry();
    public void setCountry(Country country);
}
```

Exemplo 4.7. Specify custom tuplizers in `hbm.xml`

```
<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>
```

4.6. EntityNameResolvers

`org.hibernate.EntityNameResolver` is a contract for resolving the entity name of a given entity instance. The interface defines a single method `resolveEntityName` which is passed the entity instance and is expected to return the appropriate entity name (null is allowed and would indicate that the resolver does not know how to resolve the entity name of the given entity instance). Generally speaking, an `org.hibernate.EntityNameResolver` is going to be most useful in the case of dynamic models. One example might be using proxied interfaces as your domain model. The hibernate test suite has an example of this exact style of usage under the org.hibernate.test.dynamicentity.tuplizer2. Here is some of the code from that package for illustration.

```
/**
 * A very trivial JDK Proxy InvocationHandler implementation where we proxy an
 * interface as the domain model and simply store persistent state in an internal
 * Map.  This is an extremely trivial example meant only for illustration.
 */
public final class DataProxyHandler implements InvocationHandler {
        private String entityName;
        private HashMap data = new HashMap();

        public DataProxyHandler(String entityName, Serializable id) {
                this.entityName = entityName;
                data.put( "Id", id );
        }

        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                String methodName = method.getName();
                if ( methodName.startsWith( "set" ) ) {
                        String propertyName = methodName.substring( 3 );
                        data.put( propertyName, args[0] );
                }
                else if ( methodName.startsWith( "get" ) ) {
                        String propertyName = methodName.substring( 3 );
                        return data.get( propertyName );
                }
                else if ( "toString".equals( methodName ) ) {
                        return entityName + "#" + data.get( "Id" );
                }
                else if ( "hashCode".equals( methodName ) ) {
                        return new Integer( this.hashCode() );
                }
                return null;
        }

        public String getEntityName() {
                return entityName;
        }

        public HashMap getData() {
                return data;
        }
}

public class ProxyHelper {
    public static String extractEntityName(Object object) {
        // Our custom java.lang.reflect.Proxy instances actually bundle
        // their appropriate entity name, so we simply extract it from there
        // if this represents one of our proxies; otherwise, we return null
        if ( Proxy.isProxyClass( object.getClass() ) ) {
            InvocationHandler handler = Proxy.getInvocationHandler( object );
            if ( DataProxyHandler.class.isAssignableFrom( handler.getClass() ) ) {
                DataProxyHandler myHandler = ( DataProxyHandler ) handler;
                return myHandler.getEntityName();
            }
        }
        return null;
    }

    // various other utility methods ....
```

```
}

/**
 * The EntityNameResolver implementation.
 *
 * IMPL NOTE : An EntityNameResolver really defines a strategy for how entity names
 * should be resolved.  Since this particular impl can handle resolution for all of our
 * entities we want to take advantage of the fact that SessionFactoryImpl keeps these
 * in a Set so that we only ever have one instance registered.  Why?  Well, when it
 * comes time to resolve an entity name, Hibernate must iterate over all the registered
 * resolvers.  So keeping that number down helps that process be as speedy as possible.
 * Hence the equals and hashCode implementations as is
 */
public class MyEntityNameResolver implements EntityNameResolver {
    public static final MyEntityNameResolver INSTANCE = new MyEntityNameResolver();

    public String resolveEntityName(Object entity) {
        return ProxyHelper.extractEntityName( entity );
    }

    public boolean equals(Object obj) {
        return getClass().equals( obj.getClass() );
    }

    public int hashCode() {
        return getClass().hashCode();
    }
}

public class MyEntityTuplizer extends PojoEntityTuplizer {
       public MyEntityTuplizer(EntityMetamodel entityMetamodel, PersistentClass mappedEntity) {
               super( entityMetamodel, mappedEntity );
        }

        public EntityNameResolver[] getEntityNameResolvers() {
               return new EntityNameResolver[] { MyEntityNameResolver.INSTANCE };
        }

    public String determineConcreteSubclassEntityName(Object entityInstance, SessionFactoryImplementor factory) {
        String entityName = ProxyHelper.extractEntityName( entityInstance );
        if ( entityName == null ) {
            entityName = super.determineConcreteSubclassEntityName( entityInstance, factory );
        }
        return entityName;
    }

    ...
```

Com o objetivo de registrar um `org.hibernate.EntityNameResolver`, os usuários devem tanto:

1. Implement a custom tuplizer (see *Seção 4.5, "Tuplizadores"*), implementing the `getEntityNameResolvers` method

2. Registrá-lo com o `org.hibernate.impl.SessionFactoryImpl` (que é a classe de implementação para `org.hibernate.SessionFactory`) usando o método `registerEntityNameResolver`.

Mapeamento O/R Básico

5.1. Declaração de mapeamento

Object/relational mappings can be defined in three approaches:

- using Java 5 annotations (via the Java Persistence 2 annotations)
- using JPA 2 XML deployment descriptors (described in chapter XXX)
- using the Hibernate legacy XML files approach known as hbm.xml

Annotations are split in two categories, the logical mapping annotations (describing the object model, the association between two entities etc.) and the physical mapping annotations (describing the physical schema, tables, columns, indexes, etc). We will mix annotations from both categories in the following code examples.

JPA annotations are in the `javax.persistence.*` package. Hibernate specific extensions are in `org.hibernate.annotations.*`. You favorite IDE can auto-complete annotations and their attributes for you (even without a specific "JPA" plugin, since JPA annotations are plain Java 5 annotations).

Here is an example of mapping

```
package eg;

@Entity
@Table(name="cats") @Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorValue("C") @DiscriminatorColumn(name="subclass", discriminatorType=CHAR)
public class Cat {

   @Id @GeneratedValue
   public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

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

   @Temporal(DATE) @NotNull @Column(updatable=false)
   public Date getBirthdate() { return birthdate; }
   public void setBirthdate(Date birthdate) { this.birthdate = birthdate; }
   private Date birthdate;

   @org.hibernate.annotations.Type(type="eg.types.ColorUserType")
   @NotNull @Column(updatable=false)
   public ColorType getColor() { return color; }
   public void setColor(ColorType color) { this.color = color; }
   private ColorType color;

   @NotNull @Column(updatable=false)
```

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

   @NotNull @Column(updatable=false)
   public Integer getLitterId() { return litterId; }
   public void setLitterId(Integer litterId) { this.litterId = litterId; }
   private Integer litterId;

   @ManyToOne @JoinColumn(name="mother_id", updatable=false)
   public Cat getMother() { return mother; }
   public void setMother(Cat mother) { this.mother = mother; }
   private Cat mother;

   @OneToMany(mappedBy="mother") @OrderBy("litterId")
   public Set<Cat> getKittens() { return kittens; }
   public void setKittens(Set<Cat> kittens) { this.kittens = kittens; }
   private Set<Cat> kittens = new HashSet<Cat>();
}

@Entity @DiscriminatorValue("D")
public class DomesticCat extends Cat {

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

@Entity
public class Dog { ... }
```

The legacy hbm.xml approach uses an XML schema designed to be readable and hand-editable. The mapping language is Java-centric, meaning that mappings are constructed around persistent class declarations and not table declarations.

Note que, embora muitos usuários do Hibernate escolham gravar o XML manualmente, existem diversas ferramentas para gerar o documento de mapeamento, incluindo o XDoclet Middlegen e AndroMDA.

Vamos iniciar com um exemplo de mapeamento:

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

We will now discuss the concepts of the mapping documents (both annotations and XML). We will only describe, however, the document elements and attributes that are used by Hibernate at runtime. The mapping document also contains some extra optional attributes and elements that affect the database schemas exported by the schema export tool (for example, the `not-null` attribute).

5.1.1. Entity

An entity is a regular Java object (aka POJO) which will be persisted by Hibernate.

To mark an object as an entity in annotations, use the `@Entity` annotation.

```
@Entity
public class Flight implements Serializable {
    Long id;

    @Id
    public Long getId() { return id; }

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

That's pretty much it, the rest is optional. There are however any options to tweak your entity mapping, let's explore them.

`@Table` lets you define the table the entity will be persisted into. If undefined, the table name is the unqualified class name of the entity. You can also optionally define the catalog, the schema as well as unique constraints on the table.

```
@Entity
@Table(name="TBL_FLIGHT",
       schema="AIR_COMMAND",
       uniqueConstraints=
           @UniqueConstraint(
               name="flight_number",
               columnNames={"comp_prefix", "flight_number"} ) )
public class Flight implements Serializable {
    @Column(name="comp_prefix")
    public String getCompagnyPrefix() { return companyPrefix; }

    @Column(name="flight_number")
    public String getNumber() { return number; }
}
```

The constraint name is optional (generated if left undefined). The column names composing the constraint correspond to the column names as defined before the Hibernate `NamingStrategy` is applied.

`@Entity.name` lets you define the shortcut name of the entity you can used in JP-QL and HQL queries. It defaults to the unqualified class name of the class.

Hibernate goes beyond the JPA specification and provide additional configurations. Some of them are hosted on `@org.hibernate.annotations.Entity`:

- `dynamicInsert` / `dynamicUpdate` (defaults to false): specifies that `INSERT` / `UPDATE` SQL should be generated at runtime and contain only the columns whose values are not null. The `dynamic-`

`update` and `dynamic-insert` settings are not inherited by subclasses. Although these settings can increase performance in some cases, they can actually decrease performance in others.

- `selectBeforeUpdate` (defaults to false): specifies that Hibernate should *never* perform an SQL `UPDATE` unless it is certain that an object is actually modified. Only when a transient object has been associated with a new session using `update()`, will Hibernate perform an extra SQL `SELECT` to determine if an `UPDATE` is actually required. Use of `select-before-update` will usually decrease performance. It is useful to prevent a database update trigger being called unnecessarily if you reattach a graph of detached instances to a `Session`.

- `polymorphisms` (defaults to `IMPLICIT`): determines whether implicit or explicit query polymorphisms is used. *Implicit* polymorphisms means that instances of the class will be returned by a query that names any superclass or implemented interface or class, and that instances of any subclass of the class will be returned by a query that names the class itself. *Explicit* polymorphisms means that class instances will be returned only by queries that explicitly name that class. Queries that name the class will return only instances of subclasses mapped. For most purposes, the default `polymorphisms=IMPLICIT` is appropriate. Explicit polymorphisms is useful when two different classes are mapped to the same table This allows a "lightweight" class that contains a subset of the table columns.

- `persister`: specifies a custom `ClassPersister`. The `persister` attribute lets you customize the persistence strategy used for the class. You can, for example, specify your own subclass of `org.hibernate.persister.EntityPersister`, or you can even provide a completely new implementation of the interface `org.hibernate.persister.ClassPersister` that implements, for example, persistence via stored procedure calls, serialization to flat files or LDAP. See `org.hibernate.test.CustomPersister` for a simple example of "persistence" to a `Hashtable`.

- `optimisticLock` (defaults to `VERSION`): determines the optimistic locking strategy. If you enable `dynamicUpdate`, you will have a choice of optimistic locking strategies:

  - `version`: verifica as colunas de versão/timestamp
  - `all`: verifica todas as colunas
  - `dirty`: verifica as colunas modificadas, permitindo algumas atualizações concorrentes
  - `none`: não utiliza o bloqueio otimista

  Nós *realmente* recomendamos que você utilize as colunas de versão/timestamp para o bloqueio otimista com o Hibernate. Esta é a melhor estratégia em relação ao desempenho e é a única estratégia que trata corretamente as modificações efetuadas em instâncias desconectadas (por exemplo, quando `Session.merge()` é utilizado).

Dica

Be sure to import `@javax.persistence.Entity` to mark a class as an entity. It's a common mistake to import `@org.hibernate.annotations.Entity` by accident.

Some entities are not mutable. They cannot be updated or deleted by the application. This allows Hibernate to make some minor performance optimizations.. Use the `@Immutable` annotation.

You can also alter how Hibernate deals with lazy initialization for this class. On `@Proxy`, use `lazy`=false to disable lazy fetching (not recommended). You can also specify an interface to use for lazy initializing proxies (defaults to the class itself): use `proxyClass` on `@Proxy`. Hibernate will initially return proxies (Javassist or CGLIB) that implement the named interface. The persistent object will load when a method of the proxy is invoked. See "Initializing collections and proxies" below.

`@BatchSize` specifies a "batch size" for fetching instances of this class by identifier. Not yet loaded instances are loaded batch-size at a time (default 1).

You can specific an arbitrary SQL WHERE condition to be used when retrieving objects of this class. Use `@Where` for that.

In the same vein, `@Check` lets you define an SQL expression used to generate a multi-row *check* constraint for automatic schema generation.

There is no difference between a view and a base table for a Hibernate mapping. This is transparent at the database level, although some DBMS do not support views properly, especially with updates. Sometimes you want to use a view, but you cannot create one in the database (i.e. with a legacy schema). In this case, you can map an immutable and read-only entity to a given SQL subselect expression using `@org.hibernate.annotations.Subselect`:

```
@Entity
@Subselect("select item.name, max(bid.amount), count(*) "
        + "from item "
        + "join bid on bid.item_id = item.id "
        + "group by item.name")
@Synchronize( {"item", "bid"} ) //tables impacted
public class Summary {
    @Id
    public String getId() { return id; }
    ...
}
```

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

We will now explore the same options using the hbm.xml structure. You can declare a persistent class using the `class` element. For example:

```
<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 a função é ausente, assume-se que o mapeamento é para entidades não-POJO.
2. `table` (opcional – padrão para nomes de classes não qualificadas): O nome da sua tabela do banco de dados.
3. `discriminator-value` (opcional – padrão para o nome da classe): Um valor que distingue subclasses individuais, usadas para o comportamento polimórfico. Valores aceitos incluem `null` e `not null`.
4. `mutable` (opcional - valor padrão `true`): Especifica quais instâncias da classe são (ou não) mutáveis.
5. `schema` (opcional): Sobrepõe o nome do esquema especificado pelo elemento raíz `<hibernate-mapping>`.
6. `catalog` (opcional): Sobrepõe o nome do catálogo especificado pelo elemento raíz `<hibernate-mapping>`.
7. `proxy` (opcional): Especifica uma interface para ser utilizada pelos proxies de inicialização lazy. Você pode especificar o nome da própria classe.

8. `dynamic-update` (opcional, valor padrão `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 padrão `falso`): 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 padrão `false`): Especifica que o Hibernate *nunca* deve executar um SQL de `UPDATE` a não ser que seja certo que um objeto está atualmente modificado. Em certos casos (na verdade, apenas quando um objeto transiente foi associado a uma nova sessão utilizando `update()`), isto significa que o Hibernate irá executar uma instrução SQL de `SELECT` adicional para determinar se um `UPDATE` é necessário nesse momento.
11. `polymorphisms` (optional - defaults to `implicit`): determines whether implicit or explicit query polymorphisms is used.
12. `where` (opicional): Especifica um comando SQL `WHERE` arbitrário para ser usado quando da recuperação de objetos desta classe.
13. `persister` (opcional): Especifica uma `ClassPersister` customizada.
14. `batch-size` (opcional, valor padrão `1`) Especifica um "tamanho de lote" para a recuperação de instâncias desta classe pela identificação.
15. `optimistic-lock` (opcional, valor padrão `version`): Determina a estratégia de bloqueio.
16. `lazy` (opcional): A recuperação lazy pode ser completamente desabilitada, ajustando `lazy="false"`.
17. `entity-name` (optional - defaults to the class name): Hibernate3 allows a class to be mapped multiple times, potentially to different tables. It also allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity. See *Seção 4.4, "Modelos dinâmicos"* and *Capítulo 20, Mapeamento XML* for more information.
18. `check` (opcional): Uma expressão SQL utilizada para gerar uma restrição de *verificação* de múltiplas linhas para a geração automática do esquema.
19. `rowid` (opcional): O Hibernate poder usar as então chamadas ROWIDs em bancos de dados que a suportam. Por exemplo, no Oracle, o Hibernate pode utilizar a coluna extra rowid para atualizações mais rápidas se você configurar esta opção para `rowid`. Um ROWID é uma implementação que representa de maneira detalhada a localização física de uma determinada tuple armazenada.
20. `subselect` (opcional): Mapeia uma entidade imutável e somente de leitura para um subconjunto do banco de dados. Útil se você quiser ter uma visão, ao invés de uma tabela. Veja abaixo para mais informações.
21. `abstract` (opcional): Utilizada para marcar superclasses abstratas em hierarquias `<union-subclass>`.

É perfeitamente aceitável 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 interna *estática*. Você deverá especificar o nome da classe usando a forma padrão, por exemplo: `eg.Foo$Bar`.

Here is how to do a virtual view (subselect) in XML:

```
<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>
```

The `<subselect>` is available both as an attribute and a nested mapping element.

5.1.2. Identifiers

Mapped classes *must* declare the primary key column of the database table. Most classes will also have a JavaBeans-style property holding the unique identifier of an instance.

Mark the identifier property with `@Id`.

```
@Entity
public class Person {
   @Id Integer getId() { ... }
   ...
}
```

In hbm.xml, use the `<id>` element which defines the mapping from that property to the primary key column.

```
<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 da propriedade do identificador.

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

3. `column` (opcional – padrão para o nome da propridade): O nome coluna chave primária.
4. `unsaved-value` (opcional - padrão para um valor "sensível"): O valor da propriedade de identificação que indica que a instância foi novamente instanciada (unsaved), diferenciando de instâncias desconectadas que foram salvas ou carregadas em uma sessão anterior.
5. `access` (opcional - padrão para `property`): A estratégia que o Hiberante deve utilizar para acessar o valor da propriedade.

Se a função `name` não for declarada, considera-se que a classe não tem a propriedade de identificação.

The `unsaved-value` attribute is almost never needed in Hibernate3 and indeed has no corresponding element in annotations.

You can also declare the identifier as a composite identifier. This allows access to legacy data with composite keys. Its use is strongly discouraged for anything else.

5.1.2.1. Composite identifier

You can define a composite primary key through several syntaxes:

- use a component type to represent the identifier and map it as a property in the entity: you then annotated the property as `@EmbeddedId`. The component type has to be `Serializable`.
- map multiple properties as `@Id` properties: the identifier type is then the entity class itself and needs to be `Serializable`. This approach is unfortunately not standard and only supported by Hibernate.
- map multiple properties as `@Id` properties and declare an external class to be the identifier type. This class, which needs to be `Serializable`, is declared on the entity via the `@IdClass` annotation. The identifier type must contain the same properties as the identifier properties of the entity: each property name must be the same, its type must be the same as well if the entity property is of a basic type, its type must be the type of the primary key of the associated entity if the entity property is an association (either a `@OneToOne` or a `@ManyToOne`).

As you can see the last case is far from obvious. It has been inherited from the dark ages of EJB 2 for backward compatibilities and we recommend you not to use it (for simplicity sake).

Let's explore all three cases using examples.

5.1.2.1.1. id as a property using a component type

Here is a simple example of `@EmbeddedId`.

```
@Entity
class User {
   @EmbeddedId
   @AttributeOverride(name="firstName", column=@Column(name="fld_firstname")
```

```
   UserId id;

   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;
}
```

You can notice that the `UserId` class is serializable. To override the column mapping, use `@AttributeOverride`.

An embedded id can itself contains the primary key of an associated entity.

```
@Entity
class Customer {
   @EmbeddedId CustomerId id;
   boolean preferredCustomer;

   @MapsId("userId")
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   @OneToOne User user;
}

@Embeddable
class CustomerId implements Serializable {
   UserId userId;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

In the embedded id object, the association is represented as the identifier of the associated entity. But you can link its value to a regular association in the entity via the `@MapsId` annotation. The `@MapsId` value correspond to the property name of the embedded id object containing

the associated entity's identifier. In the database, it means that the `Customer.user` and the `CustomerId.userId` properties share the same underlying column (`user_fk` in this case).

Dica

The component type used as identifier must implement `equals()` and `hashCode()`.

In practice, your code only sets the `Customer.user` property and the user id value is copied by Hibernate into the `CustomerId.userId` property.

Atenção

The id value can be copied as late as flush time, don't rely on it until after flush time.

While not supported in JPA, Hibernate lets you place your association directly in the embedded id component (instead of having to use the `@MapsId` annotation).

```
@Entity
class Customer {
   @EmbeddedId CustomerId id;
   boolean preferredCustomer;
}

@Embeddable
class CustomerId implements Serializable {
   @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;


   //implements equals and hashCode
}
```

Let's now rewrite these examples using the hbm.xml syntax.

```
<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>
```

First a simple example:

```
<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName" column="fld_firstname"/>
      <key-property name="lastName"/>
   </composite-id>
</class>
```

Then an example showing how an association can be mapped.

```
<class name="Customer">
   <composite-id name="id" class="CustomerId">
      <key-property name="firstName" column="userfirstname_fk"/>
      <key-property name="lastName" column="userfirstname_fk"/>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>

   <many-to-one name="user">
      <column name="userfirstname_fk" updatable="false" insertable="false"/>
      <column name="userlastname_fk" updatable="false" insertable="false"/>
   </many-to-one>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

Notice a few things in the previous example:

- the order of the properties (and column) matters. It must be the same between the association and the primary key of the associated entity
- the many to one uses the same columns as the primary key and thus must be marked as read only (`insertable` and `updatable` to false).
- unlike with `@MapsId`, the id value of the associated entity is not transparently copied, check the `foreign` id generator for more information.

The last example shows how to map association directly in the embedded id component.

```
<class name="Customer">
   <composite-id name="id" class="CustomerId">
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

This is the recommended approach to map composite identifier. The following options should not be considered unless some constraint are present.

5.1.2.1.2. Multiple id properties without identifier type

Another, arguably more natural, approach is to place `@Id` on multiple properties of your entity. This approach is only supported by Hibernate (not JPA compliant) but does not require an extra embeddable component.

```
@Entity
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;
```

```
   boolean preferredCustomer;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

In this case `Customer` is its own identifier representation: it must implement `Serializable` and must implement `equals()` and `hashCode()`.

In hbm.xml, the same mapping is:

```
<class name="Customer">
   <composite-id>
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

5.1.2.1.3. Multiple id properties with with a dedicated identifier type

`@IdClass` on an entity points to the class (component) representing the identifier of the class. The properties marked `@Id` on the entity must have their corresponding property on the `@IdClass`. The return type of search twin property must be either identical for basic properties or must correspond to the identifier class of the associated entity for an association.

Atenção

This approach is inherited from the EJB 2 days and we recommend against its use. But, after all it's your application and Hibernate supports it.

```
@Entity
@IdClass(CustomerId.class)
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;

   boolean preferredCustomer;
}

class CustomerId implements Serializable {
   UserId user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;

   //implements equals and hashCode
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

`Customer` and `CustomerId` do have the same properties `customerNumber` as well as `user`. `CustomerId` must be `Serializable` and implement `equals()` and `hashCode()`.

While not JPA standard, Hibernate let's you declare the vanilla associated property in the `@IdClass`.

```
@Entity
@IdClass(CustomerId.class)
```

```
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;

   boolean preferredCustomer;
}

class CustomerId implements Serializable {
   @OneToOne User user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;

   //implements equals and hashCode
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

This feature is of limited interest though as you are likely to have chosen the `@IdClass` approach to stay JPA compliant or you have a quite twisted mind.

Here are the equivalent on hbm.xml files:

```
<class name="Customer">
   <composite-id class="CustomerId" mapped="true">
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>
```

```
   <property name="age"/>
</class>
```

5.1.2.2. Identifier generator

Hibernate can generate and populate identifier values for you automatically. This is the recommended approach over "business" or "natural" id (especially composite ids).

Hibernate offers various generation strategies, let's explore the most common ones first that happens to be standardized by JPA:

- IDENTITY: supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type `long`, `short` or `int`.
- SEQUENCE (called `seqhilo` in Hibernate): uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a named database sequence.
- TABLE (called `MultipleHiLoPerTableGenerator` in Hibernate) : uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a table and column as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database.
- AUTO: selects `IDENTITY`, `SEQUENCE` or `TABLE` depending upon the capabilities of the underlying database.

Importante

We recommend all new projects to use the new enhanced identifier generators. They are deactivated by default for entities using annotations but can be activated using `hibernate.id.new_generator_mappings=true`. These new generators are more efficient and closer to the JPA 2 specification semantic.

However they are not backward compatible with existing Hibernate based application (if a sequence or a table is used for id generation). See XXXXXXX *???* for more information on how to activate them.

To mark an id property as generated, use the `@GeneratedValue` annotation. You can specify the strategy used (default to `AUTO`) by setting `strategy`.

```
@Entity
public class Customer {
   @Id @GeneratedValue
   Integer getId() { ... };
}

@Entity
public class Invoice {
```

```
   @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
   Integer getId() { ... };
}
```

`SEQUENCE` and `TABLE` require additional configurations that you can set using `@SequenceGenerator` and `@TableGenerator`:

- `name`: name of the generator
- `table` / `sequenceName`: name of the table or the sequence (defaulting respectively to `hibernate_sequences` and `hibernate_sequence`)
- `catalog` / `schema`:
- `initialValue`: the value from which the id is to start generating
- `allocationSize`: the amount to increment by when allocating id numbers from the generator

In addition, the `TABLE` strategy also let you customize:

- `pkColumnName`: the column name containing the entity identifier
- `valueColumnName`: the column name containing the identifier value
- `pkColumnValue`: the entity identifier
- `uniqueConstraints`: any potential column constraint on the table containing the ids

To link a table or sequence generator definition with an actual generated property, use the same name in both the definition `name` and the generator value `generator` as shown below.

```
@Id
@GeneratedValue(
    strategy=GenerationType.SEQUENCE,
    generator="SEQ_GEN")
@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence",
    allocationSize=20
)
public Integer getId() { ... }
```

The scope of a generator definition can be the application or the class. Class-defined generators are not visible outside the class and can override application level generators. Application level generators are defined in JPA's XML deployment descriptors (see XXXXXX *???*):

```
<table-generator name="EMP_GEN"
            table="GENERATOR_TABLE"
            pk-column-name="key"
            value-column-name="hi"
            pk-column-value="EMP"
```

```
            allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.TableGenerator(
    name="EMP_GEN",
    table="GENERATOR_TABLE",
    pkColumnName = "key",
    valueColumnName = "hi"
    pkColumnValue="EMP",
    allocationSize=20
)

<sequence-generator name="SEQ_GEN"
    sequence-name="my_sequence"
    allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence",
    allocationSize=20
)
```

If a JPA XML descriptor (like `META-INF/orm.xml`) is used to define the generators, `EMP_GEN` and `SEQ_GEN` are application level generators.

Nota

Package level definition is not supported by the JPA specification. However, you can use the `@GenericGenerator` at the package level (see *???*).

These are the four standard JPA generators. Hibernate goes beyond that and provide additional generators or additional options as we will see below. You can also write your own custom identifier generator by implementing `org.hibernate.id.IdentifierGenerator`.

To define a custom generator, use the `@GenericGenerator` annotation (and its plural counter part `@GenericGenerators`) that describes the class of the identifier generator or its short cut name (as described below) and a list of key/value parameters. When using `@GenericGenerator` and assigning it via `@GeneratedValue.generator`, the `@GeneratedValue.strategy` is ignored: leave it blank.

```
@Id @GeneratedValue(generator="system-uuid")
@GenericGenerator(name="system-uuid", strategy = "uuid")
public String getId() {

@Id @GeneratedValue(generator="trigger-generated")
@GenericGenerator(
    name="trigger-generated",
```

```
    strategy = "select",
    parameters = @Parameter(name="key", value = "socialSecurityNumber")
)
public String getId() {
```

The hbm.xml approach uses the optional `<generator>` child element inside `<id>`. If any parameters are required to configure or initialize the generator instance, they are passed using the `<param>` element.

```
<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>
```

5.1.2.2.1. Various additional generators

Todos os geradores implementam a interface `org.hibernate.id.IdentifierGenerator`. Esta é uma interface bem simples. Algumas aplicações podem prover suas próprias implementações especializadas, entretanto, o Hibernate disponibiliza um conjunto de implementações internamente. Há nomes de atalhos para estes geradores internos, conforme segue abaixo:

`increment`
: gera identificadores dos tipos `long`, `short` ou `int` que são únicos 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, Servidor MS SQL, Sybase e HypersonicSQL. O identificador retornado é do tipo `long`, `short` ou `int`.

`sequence`
: utiliza uma seqüência em DB2, PostgreSQL, Oracle, SAP DB, McKoi ou um gerador 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 padrão `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 específico.

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

`uuid`
: Generates a 128-bit UUID based on a custom algorithm. The value generated is represented as a string of 32 hexidecimal digits. Users can also configure it to use

a separator (config parameter "separator") which separates the hexidecimal digits into 8{sep}8{sep}4{sep}8{sep}4. Note specifically that this is different than the IETF RFC 4122 representation of 8-4-4-4-12. If you need RFC 4122 compliant UUIDs, consider using "uuid2" generator discussed below.

`uuid2`

Generates a IETF RFC 4122 compliant (variant 2) 128-bit UUID. The exact "version" (the RFC term) generated depends on the pluggable "generation strategy" used (see below). Capable of generating values as `java.util.UUID`, `java.lang.String` or as a byte array of length 16 (`byte[16]`). The "generation strategy" is defined by the interface `org.hibernate.id.UUIDGenerationStrategy`. The generator defines 2 configuration parameters for defining which generation strategy to use:

`uuid_gen_strategy_class`

Names the UUIDGenerationStrategy class to use

`uuid_gen_strategy`

Names the UUIDGenerationStrategy instance to use

Out of the box, comes with the following strategies:

- `org.hibernate.id.uuid.StandardRandomStrategy` (the default) - generates "version 3" (aka, "random") UUID values via the `randomUUID` method of `java.util.UUID`
- `org.hibernate.id.uuid.CustomVersionOneStrategy` - generates "version 1" UUID values, using IP address since mac address not available. If you need mac address to be used, consider leveraging one of the existing third party UUID generators which sniff out mac address and integrating it via the `org.hibernate.id.UUIDGenerationStrategy` contract. Two such libraries known at time of this writing include *http://johannburkard.de/software/uuid/* and *http://commons.apache.org/sandbox/id/uuid.html*

`guid`

utiliza um string GUID gerado pelo banco de dados no Servidor MS SQL 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 padrão caso nenhum elemento `<generator>` seja especificado.

`select`

retorna a chave primária recuperada por um trigger do banco de dados, selecionando uma linha pela chave única e recuperando o valor da chave primária.

`foreign`

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

`sequence-identity`
: uma estratégia de geração de seqüência especializada que use uma seqüência de banco de dados para a geração de valor atual, mas combina isto com JDBC3 getGeneratedKeys para de fato retornar o valor do identificador gerado como parte da execução de instrução de inserção. Esta estratégia é somente conhecida para suportar drivers da Oracle 10g, focados em JDK 1.4. Note que os comentários sobre estas instruções de inserção estão desabilitados devido a um bug nos drivers da Oracle.

5.1.2.2.2. 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 próximo 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>
```

Infelizmente, você não pode utilizar `hilo` quando estiver fornecendo sua própria `Connection` para o Hibernate. Quando o Hibernate estiver usando uma fonte de dados do servidor de aplicações para obter conexões suportadas com JTA, você precisará configurar adequadamente o `hibernate.transaction.manager_lookup_class`.

5.1.2.2.3. Algoritmo UUID

O UUID contém: o endereço IP, hora de início da JVM que é com precisão de um quarto de segundo, a hora do sistema e um valor contador que é único dentro da JVM. Não é possível obter o endereço MAC ou um endereço de memória do código Java, portanto este é o melhor que pode ser feito sem utilizar JNI.

5.1.2.2.4. Colunas de identidade e seqüências

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 `identity`, `sequence` e `hilo`, dependendo das capacidades do banco de dados utilizado.

5.1.2.2.5. Identificadores atribuídos

If you want the application to assign identifiers, as opposed to having Hibernate generate them, you can use the `assigned` generator. This special generator uses the identifier value already assigned to the object's identifier property. The generator is used when the primary key is a natural key instead of a surrogate key. This is the default behavior if you do not specify `@GeneratedValue` nor `<generator>` elements.

A escolha do gerador `assigned` faz com que o Hibernate utilize `unsaved-value="undefined"`. Isto força o Hibernate ir até o banco de dados para determinar se uma instância está transiente ou desacoplada, a não ser que haja uma versão ou uma propriedade de timestamp, ou que você definia `Interceptor.isUnsaved()`.

5.1.2.2.6. Chaves primárias geradas por triggers

O Hibernate não gera DDL com triggers, apenas para sistemas legados.

```
<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 chave substituta nomeada `person_id` cujo valor é gerado por um trigger.

5.1.2.2.7. Identity copy (foreign generator)

Finally, you can ask Hibernate to copy the identifier from another associated entity. In the Hibernate jargon, it is known as a foreign generator but the JPA mapping reads better and is encouraged.

```
@Entity
class MedicalHistory implements Serializable {
  @Id @OneToOne
  @JoinColumn(name = "person_id")
  Person patient;
}

@Entity
public class Person implements Serializable {
  @Id @GeneratedValue Integer id;
}
```

Or alternatively

```
@Entity
class MedicalHistory implements Serializable {
  @Id Integer id;

  @MapsId @OneToOne
  @JoinColumn(name = "patient_id")
  Person patient;
}

@Entity
class Person {
  @Id @GeneratedValue Integer id;
}
```

In hbm.xml use the following approach:

```
<class name="MedicalHistory">
   <id name="id">
      <generator class="foreign">
         <param name="property">patient</param>
      </generator>
   </id>
   <one-to-one name="patient" class="Person" constrained="true"/>
</class>
```

5.1.2.3. Aprimoração dos geradores de identificador

Iniciando com a liberação 3.2.3, existem dois novos geradores que representam uma reavaliação de dois diferentes aspectos da geração identificadora. O primeiro aspecto é a portabilidade

do banco de dados, o segundo é a otimização. A otimização significa que você não precisa questionar o banco de dados a cada solicitação para um novo valor de identificador. Estes dois geradores possuem por intenção substituir alguns dos geradores nomeados acima, começando em 3.3.x. No entanto, eles estão incluídos nas liberações atuais e podem ser referenciados pelo FQN.

A primeira destas novas gerações é a `org.hibernate.id.enhanced.SequenceStyleGenerator` que primeiramente é uma substituição para o gerador `sequence` e, segundo, um melhor gerador de portabilidade que o `native`. Isto é devido ao `native` normalmente escolher entre `identity` e `sequence`, que são semânticas extremamente diferentes das quais podem causar problemas súbitos em portabilidade de observação de aplicativos. No entanto, o `org.hibernate.id.enhanced.SequenceStyleGenerator` atinge a portabilidade numa maneira diferente. Ele escolhe entre uma tabela ou uma seqüência no banco de dados para armazenar seus valores de incrementação, dependendo nas capacidades do dialeto sendo usado. A diferença entre isto e o `native` é que o armazenamento baseado na tabela e seqüência possuem exatamente a mesma semântica. Na realidade, as seqüências são exatamente o que o Hibernate tenta imitar com os próprios geradores baseados na tabela. Este gerador possui um número de parâmetros de configuração:

- `sequence_name` (opcional - valor padrão `hibernate_sequence`) o nome da seqüência ou tabela a ser usada.
- `initial_value` (opcional - padrão para `1`) O valor inicial a ser restaurado a partir da seqüência/tabela. Em termos da criação de seqüência, isto é análogo à cláusula tipicamente nomeada "STARTS WITH".
- `increment_size` (opcional - padrão para `1`): o valor pelo qual as chamadas para a seqüência/tabela devem diferenciar-se. Nos termos da criação da seqüência, isto é análogo à cláusula tipicamente nomeada "INCREMENT BY".
- `force_table_use` (opcional - padrão para `false`): devemos forçar o uso de uma tabela como uma estrutura de reforço, mesmo que o dialeto possa suportar a seqüência?
- `value_column` (opcional - padrão para `next_val`): apenas relevante para estruturas de tabela, este é o nome da coluna onde na tabela que é usado para manter o valor.
- `optimizer` (optional - defaults to `none`): See *Seção 5.1.2.3.1, “Otimização do Gerador de Identificação”*

O segundo destes novos geradores é o `org.hibernate.id.enhanced.TableGenerator`, que primeiramente é uma substituição para o gerador `table`, mesmo que isto funcione muito mais como um `org.hibernate.id.MultipleHiLoPerTableGenerator`, e segundo, como uma reimplementação do `org.hibernate.id.MultipleHiLoPerTableGenerator` que utiliza a noção dos otimizadores pugláveis. Basicamente, este gerador define uma tabela capacitada de manter um número de valores de incremento simultâneo pelo uso múltiplo de filas de chaves distintas. Este gerador possui um número de parâmetros de configuração.

- `table_name` (opcional - padrão para `hibernate_sequences`): O nome da tabela a ser usado.
- `value_column_name` (opcional - padrão para `next_val`): o nome da coluna na tabela que é usado para manter o valor.

- `segment_column_name` (opcional - padrão para `sequence_name`) O nome da coluna da tabela que é usado para manter a "chave de segmento". Este é o valor que identifica qual valor de incremento a ser usado.
- `base` (opcional - padrão para `default`) O valor da "chave de segmento" para o segmento pelo qual nós queremos obter os valores de incremento para este gerador.
- `segment_value_length` (opcional - padrão para `255`): Usado para a geração do esquema. O tamanho da coluna para criar esta coluna de chave de segmento.
- `initial_value` (opcional - valor padrão para `1`): O valor inicial a ser restaurado a partir da tabela.
- `increment_size` (opcional - padrão para `1`): O valor pelo qual as chamadas subseqüentes para a tabela devem diferir-se.
- `optimizer` (optional - defaults to `??`): See *Seção 5.1.2.3.1, "Otimização do Gerador de Identificação"*.

5.1.2.3.1. Otimização do Gerador de Identificação

For identifier generators that store values in the database, it is inefficient for them to hit the database on each and every call to generate a new identifier value. Instead, you can group a bunch of them in memory and only hit the database when you have exhausted your in-memory value group. This is the role of the pluggable optimizers. Currently only the two enhanced generators (*Seção 5.1.2.3, "Aprimoração dos geradores de identificador"* support this operation.

- `none` (geralmente este é o padrão, caso nenhum otimizador for especificado): isto não executará quaisquer otimizações e alcançará o banco de dados para cada e toda solicitação.
- `hilo`: aplica-se ao algoritmo em volta dos valores restaurados do banco de dados. Espera-se que os valores a partir do banco de dados para este otimizador sejam seqüenciais. Os valores restaurados a partir da estrutura do banco de dados para este otimizador indica um "número de grupo". O `increment_size` é multiplicado pelo valor em memória para definir um grupo "hi value".
- `pooled`: assim como o caso do `hilo`, este otimizador tenta minimizar o número de tentativas no banco de dados. No entanto, nós simplesmente implementamos o valor de inicialização para o "próximo grupo" na estrutura do banco de dados ao invés do valor seqüencial na combinação com um algoritmo de agrupamento em memória. Neste caso, o `increment_size` refere-se aos valores de entrada a partir do banco de dados.

5.1.2.4. Partial identifier generation

Hibernate supports the automatic generation of some of the identifier properties. Simply use the `@GeneratedValue` annotation on one or several id properties.

Atenção

The Hibernate team has always felt such a construct as fundamentally wrong. Try hard to fix your data model before using this feature.

```
@Entity
public class CustomerInventory implements Serializable {
  @Id
  @TableGenerator(name = "inventory",
    table = "U_SEQUENCES",
    pkColumnName = "S_ID",
    valueColumnName = "S_NEXTNUM",
    pkColumnValue = "inventory",
    allocationSize = 1000)
  @GeneratedValue(strategy = GenerationType.TABLE, generator = "inventory")
  Integer id;


  @Id @ManyToOne(cascade = CascadeType.MERGE)
  Customer customer;
}

@Entity
public class Customer implements Serializable {
   @Id
   private int id;
}
```

You can also generate properties inside an `@EmbeddedId` class.

5.1.3. Optimistic locking properties (optional)

When using long transactions or conversations that span several database transactions, it is useful to store versioning data to ensure that if the same entity is updated by two conversations, the last to commit changes will be informed and not override the other conversation's work. It guarantees some isolation while still allowing for good scalability and works particularly well in read-often write-sometimes situations.

You can use two approaches: a dedicated version number or a timestamp.

A versão ou timestamp de uma propriedade nunca deve ser nula para uma instância desconectada, assim o Hibernate irá identificar qualquer instância com uma versão nula ou timestamp como transiente, não importando qual outra estratégia `unsaved-value` tenha sido especificada. *A declaração de 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 atribuídos ou chaves compostas*.

5.1.3.1. Version number

You can add optimistic locking capability to an entity using the `@Version` annotation:

```
@Entity
public class Flight implements Serializable {
...
    @Version
```

```
    @Column(name="OPTLOCK")
    public Integer getVersion() { ... }
}
```

The version property will be mapped to the `OPTLOCK` column, and the entity manager will use it to detect conflicting updates (preventing lost updates you might otherwise see with the last-commit-wins strategy).

The version column may be a numeric. Hibernate supports any kind of type provided that you define and implement the appropriate `UserVersionType`.

The application must not alter the version number set up by Hibernate in any way. To artificially increase the version number, check in Hibernate Entity Manager's reference documentation `LockModeType.OPTIMISTIC_FORCE_INCREMENT` or `LockModeType.PESSIMISTIC_FORCE_INCREMENT`.

If the version number is generated by the database (via a trigger for example), make sure to use `@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)`.

To declare a version property in hbm.xml, use:

```
<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 - tem como padrão o nome da propriedade name): O nome da coluna mantendo o número da versão.
2. `name`: O nome da propriedade da classe persistente.
3. `type` (opcional - padrão para `integer`): O tipo do número da versão.
4. `access` (opcional - padrão para `property`): A estratégia que o Hiberante deve utilizar para acessar o valor da propriedade.
5. `unsaved-value` (opcional – valor padrão para `undefined`): Um valor para a propriedade versão que indica que uma instância foi instanciada recentemente (unsaved), distinguindo de instâncias 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 padrão `never`): Especifica que este valor de propriedade da versão é na verdade gerado pelo banco de dados. Veja o *generated properties* para maiores informações.

7. `insert` (opcional - padrão para `true`): Especifica se a coluna de versão deve ser incluída na instrução de inserção do SQL. Pode ser configurado como `false` se a coluna do banco de dados estiver definida com um valor padrão de `0`.

5.1.3.2. Timestamp

Alternatively, you can use a timestamp. Timestamps are a less safe implementation of optimistic locking. However, sometimes an application might use the timestamps in other ways as well.

Simply mark a property of type `Date` or `Calendar` as `@Version`.

```
@Entity
public class Flight implements Serializable {
...
    @Version
    public Date getLastUpdate() { ... }
}
```

When using timestamp versioning you can tell Hibernate where to retrieve the timestamp value from - database or JVM - by optionally adding the `@org.hibernate.annotations.Source` annotation to the property. Possible values for the value attribute of the annotation are `org.hibernate.annotations.SourceType.VM` and `org.hibernate.annotations.SourceType.DB`. The default is `SourceType.DB` which is also used in case there is no `@Source` annotation at all.

Like in the case of version numbers, the timestamp can also be generated by the database instead of Hibernate. To do that, use `@org.hibernate.annotations.Generated(GenerationTime.ALWAYS)`.

In hbm.xml, use the `<timestamp>` element:

```
<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 - padrão para o nome da propriedade): O nome da coluna que mantém o timestamp.
2. `name`: O nome da propriedade no estilo JavaBeans do tipo `Date` ou `Timestamp` da classe persistente.

3. `access` (opcional - padrão para `property`): A estratégia que o Hiberante deve utilizar para acessar o valor da propriedade.
4. `unsaved-value` (opcional - padrão para `null`): Um valor de propriedade da versão que indica que uma instância foi recentemente instanciada (unsaved), distinguindo-a de instâncias desconectadas que foram salvas ou carregadas em sessões prévias. `Undefined` especifica que um valor de propriedade de identificação deve ser utilizado.
5. `source` (opcional - padrão para `vm`): De onde o Hibernate deve recuperar o valor timestamp? Do banco de dados ou da JVM atual? 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 os `Dialects` suportam a recuperação do carimbo de data e hora atual do banco de dados, enquanto outros podem não ser seguros para utilização em bloqueios, pela falta de precisão (Oracle 8, por exemplo).
6. `generated` (opcional - padrão para `never`): Especifica que o valor da propriedade timestamp é gerado pelo banco de dados. Veja a discussão do *generated properties* para maiores informações.

Nota

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

5.1.4. Propriedade

You need to decide which property needs to be made persistent in a given entity. This differs slightly between the annotation driven metadata and the hbm.xml files.

5.1.4.1. Property mapping with annotations

In the annotations world, every non static non transient property (field or method depending on the access type) of an entity is considered persistent, unless you annotate it as `@Transient`. Not having an annotation for your property is equivalent to the appropriate `@Basic` annotation.

The `@Basic` annotation allows you to declare the fetching strategy for a property. If set to `LAZY`, specifies that this property should be fetched lazily when the instance variable is first accessed. It requires build-time bytecode instrumentation, if your classes are not instrumented, property level lazy loading is silently ignored. The default is `EAGER`. You can also mark a property as not optional thanks to the `@Basic.optional` attribute. This will ensure that the underlying column are not nullable (if possible). Note that a better approach is to use the `@NotNull` annotation of the Bean Validation specification.

Let's look at a few examples:

```
public transient int counter; //transient property
```

```
private String firstname; //persistent property

@Transient
String getLengthInMeter() { ... } //transient property

String getName() {... } // persistent property

@Basic
int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
```

`counter`, a transient field, and `lengthInMeter`, a method annotated as `@Transient`, and will be ignored by the Hibernate. `name`, `length`, and `firstname` properties are mapped persistent and eagerly fetched (the default for simple properties). The `detailedComment` property value will be lazily fetched from the database once a lazy property of the entity is accessed for the first time. Usually you don't need to lazy simple properties (not to be confused with lazy association fetching). The recommended alternative is to use the projection capability of JP-QL (Java Persistence Query Language) or Criteria queries.

JPA support property mapping of all basic types supported by Hibernate (all basic Java types , their respective wrappers and serializable classes). Hibernate Annotations supports out of the box enum type mapping either into a ordinal column (saving the enum ordinal) or a string based column (saving the enum string representation): the persistence representation, defaulted to ordinal, can be overridden through the `@Enumerated` annotation as shown in the `note` property example.

In plain Java APIs, the temporal precision of time is not defined. When dealing with temporal data you might want to describe the expected precision in database. Temporal data can have `DATE`, `TIME`, or `TIMESTAMP` precision (ie the actual date, only the time, or both). Use the `@Temporal` annotation to fine tune that.

`@Lob` indicates that the property should be persisted in a Blob or a Clob depending on the property type: `java.sql.Clob`, `Character[]`, `char[]` and java.lang.`String` will be persisted in a Clob. `java.sql.Blob`, `Byte[]`, `byte[]` and `Serializable` type will be persisted in a Blob.

```
@Lob
public String getFullText() {
    return fullText;
}

@Lob
public byte[] getFullCode() {
    return fullCode;
```

```
}
```

If the property type implements `java.io.Serializable` and is not a basic type, and if the property is not annotated with `@Lob`, then the Hibernate `serializable` type is used.

5.1.4.1.1. Type

You can also manually specify a type using the `@org.hibernate.annotations.Type` and some parameters if needed. `@Type.type` could be:

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

If you do not specify a type, Hibernate will use reflection upon the named property and guess the correct Hibernate type. Hibernate will attempt to interpret the name of the return class of the property getter using, in order, rules 2, 3, and 4.

`@org.hibernate.annotations.TypeDef` and `@org.hibernate.annotations.TypeDefs` allows you to declare type definitions. These annotations can be placed at the class or package level. Note that these definitions are global for the session factory (even when defined at the class level). If the type is used on a single entity, you can place the definition on the entity itself. Otherwise, it is recommended to place the definition at the package level. In the example below, when Hibernate encounters a property of class `PhoneNumer`, it delegates the persistence strategy to the custom mapping type `PhoneNumberType`. However, properties belonging to other classes, too, can delegate their persistence strategy to `PhoneNumberType`, by explicitly using the `@Type` annotation.

Nota

Package level annotations are placed in a file named `package-info.java` in the appropriate package. Place your annotations before the package declaration.

```
@TypeDef(
   name = "phoneNumber",
   defaultForType = PhoneNumber.class,
   typeClass = PhoneNumberType.class
)

@Entity
public class ContactDetails {
   [...]
   private PhoneNumber localPhoneNumber;
   @Type(type="phoneNumber")
```

```
   private OverseasPhoneNumber overseasPhoneNumber;
   [...]
}
```

The following example shows the usage of the `parameters` attribute to customize the TypeDef.

```
//in org/hibernate/test/annotations/entity/package-info.java
@TypeDefs(
    {
    @TypeDef(
        name="caster",
        typeClass = CasterStringType.class,
        parameters = {
            @Parameter(name="cast", value="lower")
        }
    )
    }
)
package org.hibernate.test.annotations.entity;

//in org/hibernate/test/annotations/entity/Forest.java
public class Forest {
    @Type(type="caster")
    public String getSmallText() {
    ...
}
```

When using composite user type, you will have to express column definitions. The `@Columns` has been introduced for that purpose.

```
@Type(type="org.hibernate.test.annotations.entity.MonetaryAmountUserType")
@Columns(columns = {
    @Column(name="r_amount"),
    @Column(name="r_currency")
})
public MonetaryAmount getAmount() {
    return amount;
}


public class MonetaryAmount implements Serializable {
    private BigDecimal amount;
    private Currency currency;
    ...
}
```

5.1.4.1.2. Access type

By default the access type of a class hierarchy is defined by the position of the `@Id` or `@EmbeddedId` annotations. If these annotations are on a field, then only fields are considered for persistence and the state is accessed via the field. If there annotations are on a getter, then only the getters

are considered for persistence and the state is accessed via the getter/setter. That works well in practice and is the recommended approach.

Nota

The placement of annotations within a class hierarchy has to be consistent (either field or on property) to be able to determine the default access type. It is recommended to stick to one single annotation placement strategy throughout your whole application.

However in some situations, you need to:

- force the access type of the entity hierarchy
- override the access type of a specific entity in the class hierarchy
- override the access type of an embeddable type

The best use case is an embeddable class used by several entities that might not use the same access type. In this case it is better to force the access type at the embeddable class level.

To force the access type on a given class, use the `@Access` annotation as showed below:

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   @Embedded private Address address;
   public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Entity
public class User {
   private Long id;
   @Id public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   private Address address;
   @Embedded public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Embeddable
@Access(AcessType.PROPERTY)
public class Address {
   private String street1;
   public String getStreet1() { return street1; }
   public void setStreet1() { this.street1 = street1; }
```

```
   private hashCode; //not persistent
}
```

You can also override the access type of a single property while keeping the other properties standard.

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
   @Transient private String userId;
   @Transient private String orderId;

   @Access(AccessType.PROPERTY)
   public String getOrderNumber() { return userId + ":" + orderId; }
   public void setOrderNumber() { this.userId = ...; this.orderId = ...; }
}
```

In this example, the default access type is `FIELD` except for the `orderNumber` property. Note that the corresponding field, if any must be marked as `@Transient` or `transient`.

@org.hibernate.annotations.AccessType

The annotation `@org.hibernate.annotations.AccessType` should be considered deprecated for FIELD and PROPERTY access. It is still useful however if you need to use a custom access type.

5.1.4.1.3. Optimistic lock

It is sometimes useful to avoid increasing the version number even if a given property is dirty (particularly collections). You can do that by annotating the property (or collection) with `@OptimisticLock(excluded=true)`.

More formally, specifies that updates to this property do not require acquisition of the optimistic lock.

5.1.4.1.4. Declaring column attributes

The column(s) used for a property mapping can be defined using the `@Column` annotation. Use it to override default values (see the JPA specification for more information on the defaults). You can use this annotation at the property level for properties that are:

- not annotated at all
- annotated with `@Basic`

- annotated with `@Version`
- annotated with `@Lob`
- annotated with `@Temporal`

```
@Entity
public class Flight implements Serializable {
...
@Column(updatable = false, name = "flight_name", nullable = false, length=50)
public String getName() { ... }
```

The `name` property is mapped to the `flight_name` column, which is not nullable, has a length of 50 and is not updatable (making the property immutable).

This annotation can be applied to regular properties as well as `@Id` or `@Version` properties.

```
@Column(
    name="columnName";                                          (1)
    boolean unique() default false;                             (2)
    boolean nullable() default true;                            (3)
    boolean insertable() default true;                          (4)
    boolean updatable() default true;                           (5)
    String columnDefinition() default "";                       (6)
    String table() default "";                                  (7)
    int length() default 255;                                   (8)
    int precision() default 0; // decimal precision             (9)
    int scale() default 0; // decimal scale
```

1. `name` (optional): the column name (default to the property name)
2. `unique` (optional): set a unique constraint on this column or not (default false)
3. `nullable` (optional): set the column as nullable (default true).
4. `insertable` (optional): whether or not the column will be part of the insert statement (default true)
5. `updatable` (optional): whether or not the column will be part of the update statement (default true)
6. `columnDefinition` (optional): override the sql DDL fragment for this particular column (non portable)
7. `table` (optional): define the targeted table (default primary table)
8. `length` (optional): column length (default 255)
8. `precision` (optional): column decimal precision (default 0)
10. `scale` (optional): column decimal scale if useful (default 0)

5.1.4.1.5. Formula

Sometimes, you want the Database to do some computation for you rather than in the JVM, you might also create some kind of virtual column. You can use a SQL fragment (aka formula) instead of mapping a property into a column. This kind of property is read only (its value is calculated by your formula fragment).

```
@Formula("obj_length * obj_height * obj_width")
public long getObjectVolume()
```

The SQL fragment can be as complex as you want and even include subselects.

5.1.4.1.6. Non-annotated property defaults

If a property is not annotated, the following rules apply:

- If the property is of a single type, it is mapped as @Basic
- Otherwise, if the type of the property is annotated as @Embeddable, it is mapped as @Embedded
- Otherwise, if the type of the property is `Serializable`, it is mapped as `@Basic` in a column holding the object in its serialized version
- Otherwise, if the type of the property is `java.sql.Clob` or `java.sql.Blob`, it is mapped as `@Lob` with the appropriate `LobType`

5.1.4.2. Property mapping with hbm.xml

O elemento `<property>` declara uma propriedade de estilo JavaBean de uma classe.

```
<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 - padrão para o nome da propriedade): 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 de Hibernate.
4. `update, insert` (opcional - padrão para `true`): especifica que as colunas mapeadas devem ser incluídas nas instruções SQL de `UPDATE` e/ou `INSERT`. Ajustar ambas para `false` permite uma propriedade "derivada" pura, cujo valor é inicializado de outra propriedade, que mapeie a mesma coluna(s) ou por uma disparo ou outra aplicação.
5. `formula` (opcional): uma instrução SQL que definie o valor para uma propriedade *calculada*. Propriedades calculadas não possuem uma coluna de mapeamento para elas.
6. `access` (opcional - padrão para `property`): A estratégia que o Hiberante deve utilizar para acessar o valor da propriedade.
7. `lazy` (opcional - padrão para `false`): Especifica que esta propriedade deve ser atingida de forma lenta quando a instância da variável é acessada pela primeira vez. Isto requer instrumentação bytecode em tempo de criação.
8. `unique` (opcional): Habilita a geração de DDL de uma única restrição para as colunas. Da mesma forma, permita que isto seja o alvo de uma `property-ref`.
9. `not-null` (opcional): Habilita a geração de DDL de uma restrição de nulidade para as colunas.
10. `optimistic-lock` (opcional - padrão para `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 - padrão para `never`): Especifica que o valor da propriedade é na verdade gerado pelo banco de dados. Veja a discussão do *generated properties* para maiores informações.

typename pode ser:

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

Se você não especificar um tipo, o Hibernate irá utilizar reflexão sobre a propriedade nomeada para ter uma idéia do tipo de Hibernate correto. O Hibernate tentará interpretar o nome da classe retornada, usando as regras 2, 3 e 4 nesta ordem. Em certos casos, você ainda precisará do

atributo `type`. Por exemplo, para distinguir entre `Hibernate.DATE` e `Hibernate.TIMESTAMP`, ou para especificar um tipo customizado.

A função `access` permite que você controle como o Hibernate irá acessar a propriedade em tempo de execução. Por padrão, o Hibernate irá chamar os métodos get/set da propriedades. Se você especificar `access="field"`, o Hibernate irá bipassar os metodos get/set, acessando o campo diretamente, usando reflexão. Você pode 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 somente leitura, e o valor da propriedade é calculado em tempo de execução. Você declara este cálculo como uma expressão SQL, que traduz para cláusula `SELECT` de uma subconsulta da consulta SQL que carrega a instância:

```
<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. Isto seria o `customerId` no exemplo dado. Observe também que você pode usar o mapeamento de elemento aninhado `<formula>`, se você não gostar de usar o atributo.

5.1.5. Embedded objects (aka components)

Embeddable objects (or components) are objects whose properties are mapped to the same table as the owning entity's table. Components can, in turn, declare their own properties, components or collections

It is possible to declare an embedded component inside an entity and even override its column mapping. Component classes have to be annotated at the class level with the `@Embeddable` annotation. It is possible to override the column mapping of an embedded object for a particular entity using the `@Embedded` and `@AttributeOverride` annotation in the associated property:

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
    Address homeAddress;

    @Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="iso2", column = @Column(name="bornIso2") ),
            @AttributeOverride(name="name", column = @Column(name="bornCountryName") )
    } )
```

```
    Country bornIn;
    ...
}
```

```
@Embeddable
public class Address implements Serializable {
    String city;
    Country nationality; //no overriding here
}
```

```
@Embeddable
public class Country implements Serializable {
    private String iso2;
    @Column(name="countryName") private String name;

    public String getIso2() { return iso2; }
    public void setIso2(String iso2) { this.iso2 = iso2; }


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

An embeddable object inherits the access type of its owning entity (note that you can override that using the `@Access` annotation).

The `Person` entity has two component properties, `homeAddress` and `bornIn`. `homeAddress` property has not been annotated, but Hibernate will guess that it is a persistent component by looking for the `@Embeddable` annotation in the Address class. We also override the mapping of a column name (to `bornCountryName`) with the `@Embedded` and `@AttributeOverride` annotations for each mapped attribute of `Country`. As you can see, `Country` is also a nested component of `Address`, again using auto-detection by Hibernate and JPA defaults. Overriding columns of embedded objects of embedded objects is through dotted expressions.

```
@Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="city", column = @Column(name="fld_city") ),
            @AttributeOverride(name="nationality.iso2", column = @Column(name="nat_Iso2") ),
          @AttributeOverride(name="nationality.name", column = @Column(name="nat_CountryName") )
            //nationality columns in homeAddress are overridden
    } )
    Address homeAddress;
```

Hibernate Annotations supports something that is not explicitly supported by the JPA specification. You can annotate a embedded object with the `@MappedSuperclass` annotation to make the superclass properties persistent (see `@MappedSuperclass` for more informations).

You can also use association annotations in an embeddable object (ie `@OneToOne`, `@ManyToOne`, `@OneToMany` or `@ManyToMany`). To override the association columns you can use `@AssociationOverride`.

If you want to have the same embeddable object type twice in the same entity, the column name defaulting will not work as several embedded objects would share the same set of columns. In plain JPA, you need to override at least one set of columns. Hibernate, however, allows you to enhance the default naming mechanism through the `NamingStrategy` interface. You can write a strategy that prevent name clashing in such a situation. `DefaultComponentSafeNamingStrategy` is an example of this.

If a property of the embedded object points back to the owning entity, annotate it with the `@Parent` annotation. Hibernate will make sure this property is properly loaded with the entity reference.

In XML, use the `<component>` element.

```
<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 – padrão 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 - padrão para `property`): A estratégia que o Hiberante deve utilizar para acessar o valor da propriedade.
6. `lazy` (opcional - padrão para `false`): Especifica que este componente deve ter uma busca lazy quando a função for acessada pela primeira vez. Isto requer instrumentação bytecode de tempo de construção.

7. `optimistic-lock` (opcional – padrão para `true`): Especifica que atualizações para este componente requerem ou não aquisição de um bloqueio otimista. Em outras palavras, determina se uma versão de incremento deve ocorrer quando esta propriedade estiver suja.
8. `unique` (opcional – valor padrão `false`): Especifica que existe uma unique restrição 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.

The `<dynamic-component>` element allows a `Map` to be mapped as a component, where the property names refer to keys of the map. See *Seção 9.5, "Componentes Dinâmicos"* for more information. This feature is not supported in annotations.

5.1.6. Inheritance strategy

Java is a language supporting polymorphism: a class can inherit from another. Several strategies are possible to persist a class hierarchy:

- Single table per class hierarchy strategy: a single table hosts all the instances of a class hierarchy
- Joined subclass strategy: one table per class and subclass is present and each table persist the properties specific to a given subclass. The state of the entity is then stored in its corresponding class table and all its superclasses
- Table per class strategy: one table per concrete class and subclass is present and each table persist the properties of the class and its superclasses. The state of the entity is then stored entirely in the dedicated table for its class.

5.1.6.1. Single table per class hierarchy strategy

With this approach the properties of all the subclasses in a given mapped class hierarchy are stored in a single table.

Each subclass declares its own persistent properties and subclasses. Version and id properties are assumed to be inherited from the root class. Each subclass in a hierarchy must define a unique discriminator value. If this is not specified, the fully qualified Java class name is used.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }
```

```
@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

In hbm.xml, for the table-per-class-hierarchy mapping strategy, the `<subclass>` declaration is used. For example:

```
<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 – padrão para o nome da classe): Um valor que distingue subclasses individuais.
3. `proxy` (opcional): Especifica a classe ou interface que usará os proxies de inicialização lazy.
4. `lazy` (opcional, padrão para `true`): Configurar `lazy="false"` desabilitará o uso de inicialização lazy.

For information about inheritance mappings see *Capítulo 10, Mapeamento de Herança* .

5.1.6.1.1. Discriminador

Discriminators are required for polymorphic persistence using the table-per-class-hierarchy mapping strategy. It declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. Hibernate Core supports the follwoing restricted set of types as discriminator column: `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

Use the `@DiscriminatorColumn` to define the discriminator column as well as the discriminator type.

Nota

The enum `DiscriminatorType` used in `javax.persitence.DiscriminatorColumn` only contains the values `STRING`,

> `CHAR` and `INTEGER` which means that not all Hibernate supported types are available via the `@DiscriminatorColumn` annotation.

You can also use `@DiscriminatorFormula` to express in SQL a virtual discriminator column. This is particularly useful when the discriminator value can be extracted from one or more columns of the table. Both `@DiscriminatorColumn` and `@DiscriminatorFormula` are to be set on the root entity (once per persisted hierarchy).

`@org.hibernate.annotations.DiscriminatorOptions` allows to optionally specify Hibernate specific discriminator options which are not standardized in JPA. The available options are `force` and `insert`. The `force` attribute is useful if the table contains rows with "extra" discriminator values that are not mapped to a persistent class. This could for example occur when working with a legacy database. If `force` is set to `true` Hibernate will specify the allowed discriminator values in the `SELECT` query, even when retrieving all instances of the root class. The second option - `insert` - tells Hibernate whether or not to include the discriminator column in SQL `INSERT`s. Usually the column should be part of the `INSERT` statement, but if your discriminator column is also part of a mapped composite identifier you have to set this option to `false`.

Dica

There is also a `@org.hibernate.annotations.ForceDiscriminator` annotation which is deprecated since version 3.6. Use `@DiscriminatorOptions` instead.

Finally, use `@DiscriminatorValue` on each class of the hierarchy to specify the value stored in the discriminator column for a given entity. If you do not set `@DiscriminatorValue` on a class, the fully qualified class name is used.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

In hbm.xml, the `<discriminator>` element is used to define the discriminator column or formula:

```
<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 - padrão para `class`): O nome da coluna discriminadora.

(2) `type` (opcional - padrão para `string`): O nome que indica o tipo Hibernate.

(3) `force` (opcional - valor padrão `false`): "Força" o Hibernate a especificar valores discriminadores permitidos mesmo quando recuperando todas as instâncias da classe raíz.

(4) `insert` (opcional - valor padrão para `true`) Ajuste para `false` se sua coluna discriminadora também fizer parte do identificador composto mapeado. (Isto informa ao Hibernate para não incluir a coluna em comandos SQL `INSERT`s).

(5) `formula` (opcional): Uma expressão SQL arbitrária que é executada quando um tipo tem que ser avaliado. Permite discriminação baseada em conteúdo.

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

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

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

5.1.6.2. Joined subclass strategy

Each subclass can also be mapped to its own table. This is called the table-per-subclass mapping strategy. An inherited state is retrieved by joining with the table of the superclass. A discriminator column is not required for this mapping strategy. Each subclass must, however, declare a table column holding the object identifier. The primary key of this table is also a foreign key to the superclass table and described by the `@PrimaryKeyJoinColumn`s or the `<key>` element.

```
@Entity @Table(name="CATS")
@Inheritance(strategy=InheritanceType.JOINED)
public class Cat implements Serializable {
    @Id @GeneratedValue(generator="cat-uuid")
    @GenericGenerator(name="cat-uuid", strategy="uuid")
    String getId() { return id; }

    ...
}

@Entity @Table(name="DOMESTIC_CATS")
@PrimaryKeyJoinColumn(name="CAT")
public class DomesticCat extends Cat {
    public String getName() { return name; }
```

```
}
```

Nota

The table name still defaults to the non qualified class name. Also if `@PrimaryKeyJoinColumn` is not set, the primary key / foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass.

In hbm.xml, use the `<joined-subclass>` element. For example:

```
<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 de classe completamente qualificada da subclasse.
2. `table`: O nome da tabela da subclasse.
3. `proxy` (opcional): Especifica a classe ou interface que usará os proxies de inicialização lazy.
4. `lazy` (opcional, padrão para `true`): Configurar `lazy="false"` desabilitará o uso de inicialização lazy.

Use the `<key>` element to declare the primary key / foreign key column. The mapping at the start of the chapter would then be re-written as:

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

For information about inheritance mappings see *Capítulo 10, Mapeamento de Herança* .

5.1.6.3. Table per class strategy

A third option is to map only the concrete classes of an inheritance hierarchy to tables. This is called the table-per-concrete-class strategy. Each table defines all persistent states of the class, including the inherited state. In Hibernate, it is not necessary to explicitly map such inheritance hierarchies. You can map each class as a separate entity root. However, if you wish use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the union subclass mapping.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Flight implements Serializable { ... }
```

Or in hbm.xml:

```
<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 de classe completamente qualificada da subclasse.
2. `table`: O nome da tabela da subclasse.
3. `proxy` (opcional): Especifica a classe ou interface que usará os proxies de inicialização lazy.
4. `lazy` (opcional, padrão para `true`): Configurar `lazy="false"` desabilitará o uso de inicialização lazy.

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

For information about inheritance mappings see *Capítulo 10, Mapeamento de Herança* .

5.1.6.4. Inherit properties from superclasses

This is sometimes useful to share common properties through a technical or a business superclass without including it as a regular mapped entity (ie no specific table for this entity). For that purpose you can map them as `@MappedSuperclass`.

```
@MappedSuperclass
public class BaseEntity {
    @Basic
    @Temporal(TemporalType.TIMESTAMP)
    public Date getLastUpdate() { ... }
    public String getLastUpdater() { ... }
    ...
}

@Entity class Order extends BaseEntity {
    @Id public Integer getId() { ... }
    ...
}
```

In database, this hierarchy will be represented as an `Order` table having the `id`, `lastUpdate` and `lastUpdater` columns. The embedded superclass property mappings are copied into their entity subclasses. Remember that the embeddable superclass is not the root of the hierarchy though.

Nota

Properties from superclasses not mapped as `@MappedSuperclass` are ignored.

Nota

The default access type (field or methods) is used, unless you use the `@Access` annotation.

Nota

The same notion can be applied to `@Embeddable` objects to persist properties from their superclasses. You also need to use `@MappedSuperclass` to do that (this should not be considered as a standard EJB3 feature though)

Nota

It is allowed to mark a class as `@MappedSuperclass` in the middle of the mapped inheritance hierarchy.

Nota

Any class in the hierarchy non annotated with `@MappedSuperclass` nor `@Entity` will be ignored.

You can override columns defined in entity superclasses at the root entity level using the `@AttributeOverride` annotation.

```
@MappedSuperclass
public class FlyingObject implements Serializable {

    public int getAltitude() {
        return altitude;
    }

    @Transient
    public int getMetricAltitude() {
        return metricAltitude;
    }

    @ManyToOne
    public PropulsionType getPropulsion() {
```

```
        return metricAltitude;
    }
    ...
}

@Entity
@AttributeOverride( name="altitude", column = @Column(name="fld_altitude") )
@AssociationOverride(
   name="propulsion",
   joinColumns = @JoinColumn(name="fld_propulsion_fk")
)
public class Plane extends FlyingObject {
    ...
}
```

The `altitude` property will be persisted in an `fld_altitude` column of table `Plane` and the propulsion association will be materialized in a `fld_propulsion_fk` foreign key column.

You can define `@AttributeOverride`(s) and `@AssociationOverride`(s) on `@Entity` classes, `@MappedSuperclass` classes and properties pointing to an `@Embeddable` object.

In hbm.xml, simply map the properties of the superclass in the `<class>` element of the entity that needs to inherit them.

5.1.6.5. Mapping one entity to several tables

While not recommended for a fresh schema, some legacy databases force your to map a single entity on several tables.

Using the `@SecondaryTable` or `@SecondaryTables` class level annotations. To express that a column is in a particular table, use the `table` parameter of `@Column` or `@JoinColumn`.

```
@Entity
@Table(name="MainCat")
@SecondaryTables({
    @SecondaryTable(name="Cat1", pkJoinColumns={
        @PrimaryKeyJoinColumn(name="cat_id", referencedColumnName="id")
    ),
    @SecondaryTable(name="Cat2", uniqueConstraints={@UniqueConstraint(columnNames={"storyPart2"})})
})
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
    private String storyPart2;

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
```

```
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
}
```

In this example, `name` will be in `MainCat`. `storyPart1` will be in `Cat1` and `storyPart2` will be in `Cat2`. `Cat1` will be joined to `MainCat` using the `cat_id` as a foreign key, and `Cat2` using `id` (ie the same column name, the `MainCat` id column has). Plus a unique constraint on `storyPart2` has been set.

There is also additional tuning accessible via the `@org.hibernate.annotations.Table` annotation:

- `fetch`: If set to JOIN, the default, Hibernate will use an inner join to retrieve a secondary table defined by a class or its superclasses and an outer join for a secondary table defined by a subclass. If set to `SELECT` then Hibernate will use a sequential select for a secondary table defined on a subclass, which will be issued only if a row turns out to represent an instance of the subclass. Inner joins will still be used to retrieve a secondary defined by the class and its superclasses.
- `inverse`: If true, Hibernate will not try to insert or update the properties defined by this join. Default to false.
- `optional`: If enabled (the default), Hibernate will insert a row only if the properties defined by this join are non-null and will always use an outer join to retrieve the properties.
- `foreignKey`: defines the Foreign Key name of a secondary table pointing back to the primary table.

Make sure to use the secondary table name in the `appliesto` property

```
@Entity
@Table(name="MainCat")
@SecondaryTable(name="Cat1")
@org.hibernate.annotations.Table(
   appliesTo="Cat1",
   fetch=FetchMode.SELECT,
   optional=true)
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
    private String storyPart2;
```

```
    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
}
```

In hbm.xml, use the `<join>` element.

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

        <key ... />

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

1. `table`: O nome da tabela associada.
2. `schema` (opcional): Sobrepõe o nome do esquema especificado pelo elemento raíz `<hibernate-mapping>`.
3. `catalog` (opcional): Sobrepõe o nome do catálogo especificado pelo elemento raíz `<hibernate-mapping>`.
4. `fetch`(opcional – valor padrão `join`): Se ajustado para `join`, o padrão, o Hibernate irá usar uma união interna para restaurar um `join` definido por uma classe ou suas subclasses e uma união externa para um `join` definido por uma subclasse. Se ajustado para `select`, então o Hibernate irá usar uma seleção seqüencial para um `<join>` definida numa subclasse, que será emitido apenas se uma linha representar uma instância da subclasse. Uniões internas ainda serão utilizadas para restaurar um `<join>` definido pela classe e suas superclasses.

5. `inverse` (opcional – padrão para `false`): Se habilitado, o Hibernate não tentará inserir ou atualizar as propriedades definidas por esta união.
6. `optional` (opcional – padrão para `false`): Se habilitado, o Hibernate irá inserir uma linha apenas se as propriedades, definidas por esta junção, não forem nulas. Isto irá sempre usar uma união externa 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 fine-grained. Porém, é útil para ficar trocando entre estratégias de mapeamento de herança numa hierarquia simples, como explicaremos mais a frente.

5.1.7. Mapping one to one and one to many associations

To link one entity to an other, you need to map the association property as a to one association. In the relational model, you can either use a foreign key or an association table, or (a bit less common) share the same primary key value between the two entities.

To mark an association, use either `@ManyToOne` or `@OnetoOne`.

`@ManyToOne` and `@OneToOne` have a parameter named `targetEntity` which describes the target entity name. You usually don't need this parameter since the default value (the type of the property that stores the association) is good in almost all cases. However this is useful when you want to use interfaces as the return type instead of the regular entity.

Setting a value of the `cascade` attribute to any meaningful value other than nothing will propagate certain operations to the associated object. The meaningful values are divided into three categories.

1. basic operations, which include: `persist`, `merge`, `delete`, `save-update`, `evict`, `replicate`, `lock` and `refresh`;

2. special values: `delete-orphan` or `all` ;

3. comma-separated combinations of operation names: `cascade="persist,merge,evict"` or `cascade="all,delete-orphan"`. See *Seção 11.11, "Persistência Transitiva"* for a full explanation. Note that single valued many-to-one associations do not support orphan delete.

By default, single point associations are eagerly fetched in JPA 2. You can mark it as lazily fetched by using `@ManyToOne(fetch=FetchType.LAZY)` in which case Hibernate will proxy the association and load it when the state of the associated entity is reached. You can force Hibernate not to use a proxy by using `@LazyToOne(NO_PROXY)`. In this case, the property is fetched lazily when the instance variable is first accessed. This requires build-time bytecode instrumentation. lazy="false" specifies that the association will always be eagerly fetched.

With the default JPA options, single-ended associations are loaded with a subsequent select if set to `LAZY`, or a SQL JOIN is used for `EAGER` associations. You can however adjust the fetching strategy, ie how data is fetched by using `@Fetch`. `FetchMode` can be `SELECT` (a select is triggered when the association needs to be loaded) or `JOIN` (use a SQL JOIN to load the association while loading the owner entity). `JOIN` overrides any lazy attribute (an association loaded through a `JOIN` strategy cannot be lazy).

5.1.7.1. Using a foreign key or an association table

An ordinary association to another persistent class is declared using a

- `@ManyToOne` if several entities can point to the the target entity
- `@OneToOne` if only a single entity can point to the the target entity

and a foreign key in one table is referencing the primary key column(s) of the target table.

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}
```

The `@JoinColumn` attribute is optional, the default value(s) is the concatenation of the name of the relationship in the owner side, _ (underscore), and the name of the primary key column in the owned side. In this example `company_id` because the property name is `company` and the column id of Company is `id`.

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE}, targetEntity=CompanyImpl.class )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
```

```
    }
    ...
}

public interface Company {
    ...
}
```

You can also map a to one association through an association table. This association table described by the `@JoinTable` annotation will contains a foreign key referencing back the entity table (through `@JoinTable.joinColumns`) and a a foreign key referencing the target entity table (through `@JoinTable.inverseJoinColumns`).

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinTable(name="Flight_Company",
        joinColumns = @JoinColumn(name="FLIGHT_ID"),
        inverseJoinColumns = @JoinColumn(name="COMP_ID")
    )
    public Company getCompany() {
        return company;
    }
    ...
}
```

Nota

You can use a SQL fragment to simulate a physical join column using the `@JoinColumnOrFormula` / `@JoinColumnOrformulas` annotations (just like you can use a SQL fragment to simulate a property column via the `@Formula` annotation).

```
@Entity
public class Ticket implements Serializable {
    @ManyToOne
    @JoinColumnOrFormula(formula="(firstname + ' ' + lastname)")
    public Person getOwner() {
        return person;
    }
    ...
}
```

You can mark an association as mandatory by using the `optional=false` attribute. We recommend to use Bean Validation's `@NotNull` annotation as a better alternative however. As a consequence, the foreign key column(s) will be marked as not nullable (if possible).

When Hibernate cannot resolve the association because the expected associated element is not in database (wrong id on the association column), an exception is raised. This might be inconvenient for legacy and badly maintained schemas. You can ask Hibernate to ignore such elements instead of raising an exception using the `@NotFound` annotation.

Exemplo 5.1. @NotFound annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @NotFound(action=NotFoundAction.IGNORE)
    public Parent getParent() { ... }
    ...
}
```

Sometimes you want to delegate to your database the deletion of cascade when a given entity is deleted. In this case Hibernate generates a cascade delete constraint at the database level.

Exemplo 5.2. @OnDelete annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @OnDelete(action=OnDeleteAction.CASCADE)
    public Parent getParent() { ... }
    ...
}
```

Foreign key constraints, while generated by Hibernate, have a fairly unreadable name. You can override the constraint name using `@ForeignKey`.

Exemplo 5.3. @ForeignKey annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @ForeignKey(name="FK_PARENT")
    public Parent getParent() { ... }
    ...
}

alter table Child add constraint FK_PARENT foreign key (parent_id) references Parent
```

Sometimes, you want to link one entity to an other not by the target entity primary key but by a different unique key. You can achieve that by referencing the unique key column(s) in `@JoinColumn.referenceColumnName`.

```
@Entity
class Person {
   @Id Integer personNumber;
   String firstName;
   @Column(name="I")
   String initial;
   String lastName;
}

@Entity
class Home {
   @ManyToOne
   @JoinColumns({
      @JoinColumn(name="first_name", referencedColumnName="firstName"),
      @JoinColumn(name="init", referencedColumnName="I"),
      @JoinColumn(name="last_name", referencedColumnName="lastName"),
   })
   Person owner
}
```

This is not encouraged however and should be reserved to legacy mappings.

In hbm.xml, mapping an association is similar. The main difference is that a `@OneToOne` is mapped as `<many-to-one unique="true"/>`, let's dive into the subject.

```
<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 da chave exterior. Isto pode também ser especificado através de elementos aninhados `<column>`.
3. `class` (opcional – padrão para o tipo de propriedade determinado pela reflexão): O nome da classe associada.
4. `cascade` (opcional): Especifica qual operação deve ser cascateada do objeto pai para o objeto associado.
5. `fetch` (opcional - padrão para `select`): Escolhe entre recuperação da união exterior ou recuperação seqüencial de seleção.
6. `update, insert` (opcional - valor padrão `true`): especifica que as colunas mapeadas devem ser incluídas em instruções SQL de `UPDATE` e/ou `INSERT`. Com o ajuste de ambas para `false` você permite uma associação "derivada" pura cujos valores são inicializados de algumas outras propriedades que mapeiam a(s) mesma(s) coluna(s) ou por um trigger ou outra aplicação.
7. `property-ref`: (opcional) O nome de uma propriedade da classe associada que esteja unida à esta chave exterior. Se não for especificada, a chave primária da classe associada será utilizada.
8. `access` (opcional - padrão para `property`): A estratégia que o Hiberante deve utilizar para acessar o valor da propriedade.
9. `unique` (opcional): Habilita a geração DDL de uma restrição única para a coluna da chave exterior. Além disso, permite ser o alvo de uma `property-ref`. Isso torna a multiplicidade da associação efetivamente um para um.
10. `not-null` (opcional): Habilita a geração DDL de uma restrição de nulidade para as colunas de chaves exteriores.
11. `optimistic-lock` (opcional - padrão para `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.
12. `lazy`(opcional – padrão para `proxy`): Por padrão, associações de ponto único são envoltas em um proxie. `lazy="no-proxy"` especifica que a propriedade deve ser trazida de forma tardia quando a instância da variável é acessada pela primeira vez. Isto requer instrumentação bytecode em tempo de criação. O `lazy="false"` especifica que a associação será sempre procurada.
13. `not-found` (opcional - padrão para `exception`): Especifica como as chaves exteriores que informam que linhas que estejam faltando serão manuseadas. O `ignore` tratará a linha faltante como uma associação nula.
14. `entity-name` (opcional): O nome da entidade da classe associada.
15. `formula` (optional): Uma instrução SQL que define um valor para uma chave exterior *computed*.

Setting a value of the `cascade` attribute to any meaningful value other than `none` will propagate certain operations to the associated object. The meaningful values are divided into three categories. First, basic operations, which include: `persist, merge, delete, save-update, evict, replicate, lock and refresh`; second, special values: `delete-orphan`; and third,`all` comma-separated combinations of operation names: `cascade="persist,merge,evict"` or `cascade="all,delete-orphan"`. See *Seção 11.11, "Persistência Transitiva"* for a full explanation. Note that single valued, many-to-one and one-to-one, associations do not support orphan delete.

Segue abaixo uma amostra de uma típica declaração `many-to-one`:

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

O atributo `property-ref` deve apenas ser usado para mapear dados legados onde uma chave exterior se refere à uma chave exclusiva da tabela associada que não seja a chave primária. Este é um modelo relacional desagradável. Por exemplo, suponha que a classe `Product` tenha um número seqüencial exclusivo, que não seja a chave primária. 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"/>
```

No entanto, isto não é recomendável.

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"/>
```

5.1.7.2. Sharing the primary key with the associated entity

The second approach is to ensure an entity and its associated entity share the same primary key. In this case the primary key column is also a foreign key and there is no extra column. These associations are always one to one.

Exemplo 5.4. One to One association

```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }

    @OneToOne(cascade = CascadeType.ALL)
    @MapsId
    public Heart getHeart() {
        return heart;
    }
    ...
}

@Entity
public class Heart {
    @Id
    public Long getId() { ...}
}
```

Nota

Many people got confused by these primary key based one to one associations. They can only be lazily loaded if Hibernate knows that the other side of the association is always present. To indicate to Hibernate that it is the case, use `@OneToOne(optional=false)`.

In hbm.xml, use the following mapping.

```
<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 – padrão para o tipo de propriedade determinado pela reflexão): 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 restrição de chave exterior na chave primária da tabela mapeada referencia a tabela da classe associada. Esta opção afeta a ordem em que `save()` e `delete()` são cascateadas, e determina se a associação pode sofrer o proxie. Isto também é usado pela ferramenta schema export.
5. `fetch` (opcional - padrão para `select`): Escolhe entre recuperação da união exterior ou recuperação seqüencial de seleção.
6. `property-ref`(opcional): O nome da propriedade da classe associada que é ligada à chave primária desta classe. Se não for especificada, a chave primária da classe associada é utilizada.
7. `access` (opcional - padrão para `property`): A estratégia que o Hiberante deve utilizar para acessar o valor da propriedade.
8. `formula` (opcional): Quase todas associações um-pra-um mapeiam para a chave primária da entidade dona. Caso este não seja o caso, você pode especificar uma outra coluna, colunas ou expressões para unir utilizando uma fórmula SQL. Veja `org.hibernate.test.onetooneformula` para exemplo.
9. `lazy` (opcional – valor padrão `proxy`): Por padrão, as associações de ponto único estão em proxy. `lazy="no-proxy"` especifica que a propriedade deve ser recuperada de forma preguiçosa quando a variável da instância for acessada pela primeira vez. Isto requer instrumentação de bytecode de tempo de construção. `lazy="false"` especifica que a associação terá sempre uma busca antecipada (eager fetched). *Note que se `constrained="false"`, será impossível efetuar o proxing e o Hibernate irá realizar uma busca antecipada na associação*.
10. `entity-name` (opcional): O nome da entidade da classe associada.

Associações de chave primária não necessitam de uma coluna extra de tabela. Se duas linhas forem relacionadas pela associação, então as duas linhas da tabela dividem o mesmo valor da chave primária. Assim, se você quiser que dois objetos sejam relacionados por uma associação de chave primária, você deve ter certeza que foram atribuídos com o mesmo valor identificador.

Para uma associação de chave primária, 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 devemos assegurar que as chaves primárias de 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` é atribuída com o mesmo valor da chave primária da instância de `Employee` referenciada com a propriedade `employee` daquela `Person`.

5.1.8. Id Natural

Although we recommend the use of surrogate keys as primary keys, you should try to identify natural keys for all entities. A natural key is a property or combination of properties that is unique and non-null. It is also immutable. Map the properties of the natural key as `@NaturalId` or map them inside the `<natural-id>` element. Hibernate will generate the necessary unique key and nullability constraints and, as a result, your mapping will be more self-documenting.

```
@Entity
public class Citizen {
    @Id
    @GeneratedValue
    private Integer id;
    private String firstname;
    private String lastname;

    @NaturalId
    @ManyToOne
    private State state;

    @NaturalId
    private String ssn;
    ...
}



//and later on query
List results = s.createCriteria( Citizen.class )
                .add( Restrictions.naturalId().set( "ssn", "1234" ).set( "state", ste ) )
                .list();
```

Or in XML,

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

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

Este mapeamento não pretende ser utilizado com entidades com chaves naturais primárias.

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

5.1.9. Any

There is one more type of property mapping. The `@Any` mapping defines a polymorphic association to classes from multiple tables. This type of mapping requires more than one column. The first column contains the type of the associated entity. The remaining columns contain the identifier. It is impossible to specify a foreign key constraint for this kind of association. This is not the usual way of mapping polymorphic associations and you should use this only in special cases. For example, for audit logs, user session data, etc.

The `@Any` annotation describes the column holding the metadata information. To link the value of the metadata information and an actual entity type, The `@AnyDef` and `@AnyDefs` annotations are used. The `metaType` attribute allows the application to specify a custom type that maps database column values to persistent classes that have identifier properties of the type specified by `idType`. You must specify the mapping from values of the `metaType` to class names.

```
@Any( metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
@AnyMetaDef(
    idType = "integer",
    metaType = "string",
    metaValues = {
        @MetaValue( value = "S", targetEntity = StringProperty.class ),
        @MetaValue( value = "I", targetEntity = IntegerProperty.class )
    } )
@JoinColumn( name = "property_id" )
public Property getMainProperty() {
    return mainProperty;
}
```

Note that `@AnyDef` can be mutualized and reused. It is recommended to place it as a package metadata in this case.

```
//on a package
@AnyMetaDef( name="property"
    idType = "integer",
    metaType = "string",
    metaValues = {
        @MetaValue( value = "S", targetEntity = StringProperty.class ),
        @MetaValue( value = "I", targetEntity = IntegerProperty.class )
    } )
package org.hibernate.test.annotations.any;


//in a class
    @Any( metaDef="property", metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
    @JoinColumn( name = "property_id" )
    public Property getMainProperty() {
        return mainProperty;
    }
```

The hbm.xml equivalent is:

```
<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>
```

Nota

You cannot mutualize the metadata in hbm.xml as you can in annotations.

```
<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 – padrão para `string`): Qualquer tipo que é permitido para um mapeamento discriminador.
4. `cascade` (opcional – valor padrão `none`): o estilo cascata.
5. `access` (opcional - padrão para `property`): A estratégia que o Hiberante deve utilizar para acessar o valor da propriedade.
6. `optimistic-lock` (opcional - valor padrão`true`): Especifica que as atualizações para esta propriedade requerem ou não aquisição da bloqueio otimista. Em outras palavras, define se uma versão de incremento deve ocorrer se esta propriedade for suja.

5.1.10. Propriedades

O elemento `<properties>` permite a definição de um grupo com nome, lógico de propriedades de uma classe. A função mais importante do construtor é que ele permite que a combinação de propriedades seja o objetivo de uma `property-ref`. É também um modo conveninente para definir uma restrição única de múltiplas colunas. Por exemplo:

```
<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. Isto *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 – padrão para `true`): Especifica que atualizações para estes componentes requerem ou não aquisição de um bloqueio otimista. Em outras palavras, determina se uma versão de incremento deve ocorrer quando estas propriedades estiverem sujas.
5. `unique` (opcional – valor padrão `false`): Especifica que existe uma unique restrição em todas as colunas mapeadas do componente.

Por exemplo, se 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 podemos ter uma associação de dados legados que referem a esta chave exclusiva da tabela `Person`, ao invés de se referirem a chave primária:

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

Nota

When using annotations as a mapping strategy, such construct is not necessary as the binding between a column and its related column on the associated table is done directly

```
@Entity
class Person {
   @Id Integer personNumber;
   String firstName;
   @Column(name="I")
   String initial;
   String lastName;
}

@Entity
class Home {
   @ManyToOne
   @JoinColumns({
      @JoinColumn(name="first_name", referencedColumnName="firstName"),
      @JoinColumn(name="init", referencedColumnName="I"),
      @JoinColumn(name="last_name", referencedColumnName="lastName"),
   })
   Person owner
}
```

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

5.1.11. Some hbm.xml specificities

The hbm.xml structure has some specificities naturally not present when using annotations, let's describe them briefly.

5.1.11.1. 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 seu DTD com o conteúdo do seu classpath.

5.1.11.1.1. Solucionador de Entidade

O Hibernate irá primeiro tentar solucionar os DTDs em seus classpath. Isto é feito, registrando uma implementação `org.xml.sax.EntityResolver` personalizada com o SAXReader que ele utiliza para ler os arquivos xml. Este `EntityResolver` personalizado, reconhece dois nomes de espaço de sistemas Id diferentes:

- a `hibernate namespace` is recognized whenever the resolver encounters a systemId starting with `http://www.hibernate.org/dtd/`. The resolver attempts to resolve these entities via the classloader which loaded the Hibernate classes.
- Um `user namespace` é reconhecido quando um solucionador encontra um sistema Id, utilizando um protocolo URL de `classpath://`. O solucionador tentará solucionar estas entidades através do carregador de classe do contexto de thread atual (1) e o carregador de classe (2) que carregou as classes do Hibernate.

Um exemplo de utilização do espaço de nome do 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 `your.domain` e contém um *typedef* personalizado.

5.1.11.2. Mapeamento do Hibernate

Este elemento possui diversos atributos opcionais. Os atributos `schema` e `catalog` especificam que tabelas referenciadas neste mapeamento pertencem ao esquema e/ou ao catálogo nomeado. Se especificados, os nomes das tabelas serão qualificados no esquema ou catálogo dado. Se não, os nomes das tabelas não serão qualificados. O atributo `default-cascade` especifica qual estilo de cascata será considerado pelas propriedades e coleções que não especificarem uma função `cascade`. A função `auto-import` nos deixa utilizar nomes de classes não qualificados na linguagem de consulta, por padrão.

```
<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 – o padrão é `none`): Um estilo cascata padrão.
4. `default-access` (opcional – o padrão é `property`): A estratégia que o Hibernate deve utilizar para acessar todas as propridades. Pode ser uma implementação personalizada de `PropertyAccessor`.
5. `default-lazy` (opcional - o padrão é `true`): O valor padrão para atributos `lazy` não especificados da classe e dos mapeamentos de coleções.
6. `auto-import` (opcional - o padrão é `true`): Especifica se podemos usar nomes de classes não qualificados, das classes deste mapeamento, na linguagem de consulta.
7. `package` (opcional): Especifica um prefixo do pacote a ser considerado para nomes de classes não qualificadas no documento de mapeamento.

Se você tem duas classes persistentes com o mesmo nome (não qualificadas), você deve ajustar `auto-import="false"`. Caso você tentar ajustar duas classes para o mesmo nome "importado", isto resultará numa exceção.

Observe que o elemento `hibernate-mapping` permite que você aninhe 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 nomeá-la após a superclasse persistente, por exemplo: `Cat.hbm.xml`, `Dog.hbm.xml`, ou se estiver usando herança, `Animal.hbm.xml`.

5.1.11.3. Key

The `<key>` element is featured a few times within this guide. It appears anywhere the parent mapping element defines a join to a new table that references the primary key of the original table. It also defines the foreign key in the joined table:

```
<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 da chave exterior. Isto pode também ser especificado através de elementos aninhados `<column>`.

(2) `on-delete` (opcional, padrão para `noaction`): Especifica se a restrição da chave exterior no banco de dados está habilitada para o deletar cascade.

(3) `property-ref` (opcional): Especifica que a chave exterior se refere a colunas que não são chave primária da tabela original. Útil para os dados legados.

(4) `not-null` (opcional): Especifica que a coluna da chave exterior não aceita valores nulos. Isto é implícito em qualquer momento que a chave exterior também fizer parte da chave primária.

(5) `update` (opcional): Especifica que a chave exterior nunca deve ser atualizada. Isto está implícito em qualquer momento que a chave exterior também fizer parte da chave primária.

(6) `unique` (opcional): Especifica que a chave exterior deve ter uma restrição única. Isto é, implícito em qualquer momento que a chave exterior também fizer parte da chave primária.

Nós recomendamos que para sistemas que o desempenho deletar seja importante, todas as chaves devem ser definidas `on-delete="cascade"`. O Hibernate irá usar uma restrição 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 bloqueio otimista do Hibernate para dados versionados.

As funções `not-null` e `update` são úteis quando estamos mapeando uma associação unidirecional um para muitos. Se você mapear uma associação unidirecional um para muitos para uma chave exterior não-nula, você *deve* declarar a coluna chave usando `<key not-null="true">`.

5.1.11.4. Importar

Vamos supor que a sua aplicação tenha duas classes persistentes com o mesmo nome, e você não quer especificar o nome qualificado do pacote nas consultas do Hibernate. As Classes deverão 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 – padrão para o nome da classe não qualificada): Um nome que pode ser usado numa linguagem de consulta.

Nota

This feature is unique to hbm.xml and is not supported in annotations.

5.1.11.5. Elementos coluna e fórmula

Qualquer elemento de mapeamento que aceita uma função `column` irá aceitar alternativamente um sub-elemento `<column>`. Da mesma forma, `<formula>` é uma alternativa para a função `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"
        read="SQL expression"
        write="SQL expression"/>
```

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

A maioria das funções no `column` fornecem um significado de junção do DDL durante a geração automática do esquema. As funções `read` e `write` permitem que você especifique o SQL personalizado, do qual o Hibernate usará para acessar o valor da coluna. Consulte a discussão da *column read and write expressions* para maiores informações.

Os elementos `column` e `formula` podem até ser combinados dentro da mesma propriedade ou associação mapeando para expressar, por exemplo, condições de 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>
```

5.2. Tipos do Hibernate

5.2.1. Entidades e valores

Os objetos de nível de linguagem Java são classificados em dois grupos, em relação ao serviço de persistência:

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 coletor de lixo. Entidades devem ser explicitamente salvas ou deletadas (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 se refere mais à forma como os objetos de aplicações são geralmente usados em grandes sistemas. Entidades suportam referências circulares e comuns. Eles podem ser versionados.

O estado persistente da entidade 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 de valor (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, estivemos usando o termo "classe persistente" para referir às entidades. Continuaremos a fazer isto. No entanto, 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 semântico. Dada esta definição, nós podemos dizer que todos os tipos (classes) fornecidos pelo JDK têm tipo de valor semântico em Java, enquanto que tipos definidos pelo usuário, podem ser

mapeados com entidade ou valor de tipo semântico. Esta decisão pertence ao desenvolvedor da aplicação. Uma boa dica para uma classe de 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 tipo de valor.

Iremos rever ambos os conceitos durante todo o guia de referência.

O desafio é 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 fornecida pelo Hibernate. Para entidades que usam `<class>`, `< subclass>` e assim por diante. Para tipos de valores nós usamos `<property>`, `<component>`, etc, geralmente com uma função `type`. O valor desta função é o nome de um *tipo de mapeamento* do Hibernate. O Hibernate fornece muitos mapeamentos imediatos para tipos de valores do JDK padrão. Você pode escrever os seus próprios tipos de mapeamentos e implementar sua estratégia de conversão adaptada, como você.

Todos os tipos internos do hibernate exceto coleções, suportam semânticas nulas com a exceção das coleções.

5.2.2. Valores de tipos básicos

Os *tipos de mapeamento básicos* fazem parte da categorização do seguinte:

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`
: Tipos de mapeamentos de classes primitivas ou wrapper Java específicos (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` and `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 matrizes de bytes para um tipo binário de SQL apropriado.

`text`
: Mapeia strings de Java longos 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` and `java.sql.Blob`. Estes tipos podem ser inconvenientes para algumas aplicações, visto que o objeto blob ou clob não pode 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`
: Mapeamento de tipos para, os geralmente considerados, tipos mutáveis de Java. Isto é 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. Leia abaixo para maiores informações.

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 criarem seus próprios tipos de valores. 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 pode 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 outras funcionalidades.

```
<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 mesmo pode 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 `padrão` 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 de parâmetro padrão 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"/>
```

Também é possível substituir os parâmetros fornecidos em um tipo de definição em situações de caso a caso, utilizando tipos de parâmetros no mapeamento da propriedade.

Apesar da rica variedade, os tipos construídos do Hibernate e suporte para componentes raramente irão utilizar um tipo de padrão, no entanto, é considerado uma boa idéia, utilizar tipos customizados para classes não entidade que ocorrem com freqüência em seu aplicativo. Por exemplo, uma classe `MonetaryAmount` é um bom candidato para um `CompositeUserType`, apesar de poder ter sido mapeado facilmente como um componente. Uma motivação para isto é

a abstração. Com um tipo padronizado, seus documentos de mapeamento seriam colocados à prova contra mudanças possíveis na forma de representação de valores monetários.

5.3. Mapeando uma classe mais de uma vez

É possível fornecer mais de um mapeamento para uma classe persistente em específico. Neste caso, você deve especificar um *nome de entidade* para as instâncias das duas entidades mapeadas não se tornarem ambíguas. Por padrão, o nome da entidade é o mesmo do nome da classe. O Hibernate o deixa especificar o nome de entidade quando estiver trabalhando com objetos persistentes, quando escrever consultas, ou ao mapear associações para a entidade nomeada.

```
<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>
```

Note como as associações são agora especificadas utilizando o `entity-name` ao invés da `class`.

Nota

This feature is not supported in Annotations

5.4. Identificadores quotados do SQL

Você poderá forçar o Hibernate a quotar um identificador no SQL gerado, anexando o nome da tabela ou coluna aos backticks no documento de mapeamento. O Hibernate usará o estilo de quotação correto para o SQL `Dialect`. Geralmente são quotas dúplas, mas parênteses para o Servidor SQL e backticks para MeuSQL.

```
@Entity @Table(name="`Line Item`")
class LineItem {
   @id @Column(name="`Item Id`") Integer id;
   @Column(name="`Item #`") int itemNumber
```

```
}

<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.5. Propriedades geradas

Propriedades Geradas são propriedades que possuem seus valores gerados pelo banco de dados. Como sempre, os aplicativos do Hibernate precisavam `renovar` objetos que contenham qualquer propriedade para qual o banco de dados estivesse gerando valores. No entanto, vamos permitir que o aplicativo delegue esta responsabilidade ao Hibernate. Essencialmente, quando o Hibernate edita um SQL INSERT ou UPDATE para uma entidade que tem propriedades geradas definidas, ele edita imediatamente depois uma seleção para recuperar os valores gerados.

As propriedades marcadas como geradas devem ser não-inseríveis e não-atualizáveis. Somente *versions*, *timestamps*, e *simple properties* podem ser marcadas como geradas.

`never` (padrão) - significa que o valor de propriedade dado não é gerado dentro do banco de dados.

`insert`: informa que o valor de propriedade dado é gerado ao inserir, mas não é novamente gerado nas próximas atualizações. Propriedades do tipo data criada, se encaixam nesta categoria. Note que embora as propriedades *version* e *timestamp* podem ser marcadas como geradas, esta opção não está disponível.

`always` - informa que o valor da propriedade é gerado tanto ao inserir quanto ao atualizar.

To mark a property as generated, use `@Generated`.

5.6. Column transformers: read and write expressions

Hibernate allows you to customize the SQL it uses to read and write the values of columns mapped to *simple properties*. For example, if your database provides a set of data encryption functions, you can invoke them for individual columns like this:

```
@Entity
class CreditCard {
   @Column(name="credit_card_num")
   @ColumnTransformer(
      read="decrypt(credit_card_num)",
      write="encrypt(?)")
   public String getCreditCardNumber() { return creditCardNumber; }
   public void setCreditCardNumber(String number) { this.creditCardNumber = number; }
   private String creditCardNumber;
}
```

or in XML

```
<property name="creditCardNumber">
        <column
          name="credit_card_num"
          read="decrypt(credit_card_num)"
          write="encrypt(?)"/>
</property>
```

Nota

You can use the plural form `@ColumnTransformers` if more than one columns need to define either of these rules.

If a property uses more that one column, you must use the `forColumn` attribute to specify which column, the expressions are targeting.

```
@Entity
class User {
   @Type(type="com.acme.type.CreditCardType")
   @Columns( {
      @Column(name="credit_card_num"),
      @Column(name="exp_date") } )
   @ColumnTransformer(
      forColumn="credit_card_num",
      read="decrypt(credit_card_num)",
      write="encrypt(?)")
   public CreditCard getCreditCard() { return creditCard; }
   public void setCreditCard(CreditCard card) { this.creditCard = card; }
   private CreditCard creditCard;
}
```

O Hibernate aplica automaticamente as expressöes personalizadas a todo instante que a propriedade é referenciada numa consulta. Esta funcionalidade é parecida a uma `formula` de propriedade-derivada com duas diferenças:

- Esta propriedade é suportada por uma ou mais colunas que são exportadas como parte da geração do esquema automático.
- Esta propriedade é de gravação-leitura, e não de leitura apenas.

Caso a expressão `write`seja especificada, deverá conter um '?' para o valor.

5.7. Objetos de Banco de Dados Auxiliares

Permite o uso dos comandos CREATE e DROP para criar e remover os objetos de banco de dados arbitrários. Juntamente às ferramentas de evolução do esquema do Hibernate, eles

possuem a habilidade de definir completamente um esquema de usuário dentro dos arquivos de mapeamento do Hibernate. Embora criado especificamente para criar e remover algo como trigger ou procedimento armazenado, qualquer comando SQL que pode rodar através de um método `java.sql.Statement.execute()` é válido. Existem dois módulos essenciais para definir objetos de banco de dados auxiliares:

O primeiro módulo é para listar explicitamente os comandos CREATE e DROP 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 módulo é para fornecer uma classe padrão que sabe como construir os comandos CREATE e DROP. Esta classe padrão deve implementar a interface `org.hibernate.mapping.AuxiliaryDatabaseObject`.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
</hibernate-mapping>
```

Além disso, estes objetos de banco de dados podem ter um escopo opcional que só será aplicado quando certos dialetos forem utilizados.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9iDialect"/>
        <dialect-scope name="org.hibernate.dialect.Oracle10gDialect"/>
    </database-object>
</hibernate-mapping>
```

Nota

This feature is not supported in Annotations

Types

As an Object/Relational Mapping solution, Hibernate deals with both the Java and JDBC representations of application data. An online catalog application, for example, most likely has `Product` object with a number of attributes such as a `sku`, `name`, etc. For these individual attributes, Hibernate must be able to read the values out of the database and write them back. This 'marshalling' is the function of a *Hibernate type*, which is an implementation of the `org.hibernate.type.Type` interface. In addition, a *Hibernate type* describes various aspects of behavior of the Java type such as "how is equality checked?" or "how are values cloned?".

Importante

A Hibernate type is neither a Java type nor a SQL datatype; it provides a information about both.

When you encounter the term *type* in regards to Hibernate be aware that usage might refer to the Java type, the SQL/JDBC type or the Hibernate type.

Hibernate categorizes types into two high-level groups: value types (see *Seção 6.1, "Value types"*) and entity types (see *Seção 6.2, "Entity types"*).

6.1. Value types

The main distinguishing characteristic of a value type is the fact that they do not define their own lifecycle. We say that they are "owned" by something else (specifically an entity, as we will see later) which defines their lifecycle. Value types are further classified into 3 sub-categories: basic types (see *Seção 6.1.1, "Basic value types"*), composite types (see *Seção 6.1.2, "Composite types"*) amd collection types (see *Seção 6.1.3, "Collection types"*).

6.1.1. Basic value types

The norm for basic value types is that they map a single database value (column) to a single, non-aggregated Java type. Hibernate provides a number of built-in basic types, which we will present in the following sections by the Java type. Mainly these follow the natural mappings recommended in the JDBC specification. We will later cover how to override these mapping and how to provide and use alternative type mappings.

6.1.1.1. java.lang.String

`org.hibernate.type.StringType`

Maps a string to the JDBC VARCHAR type. This is the standard mapping for a string if no Hibernate type is specified.

Registered under `string` and `java.lang.String` in the type registry (see *Seção 6.5, "Type registry"*).

`org.hibernate.type.MaterializedClob`
: Maps a string to a JDBC CLOB type

  Registered under `materialized_clob` in the type registry (see *Seção 6.5, "Type registry"*).

`org.hibernate.type.TextType`
: Maps a string to a JDBC LONGVARCHAR type

  Registered under `text` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.2. `java.lang.Character` (or char primitive)

`org.hibernate.type.CharacterType`
: Maps a char or `java.lang.Character` to a JDBC CHAR

  Registered under `char` and `java.lang.Character` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.3. `java.lang.Boolean` (or boolean primitive)

`org.hibernate.type.BooleanType`
: Maps a boolean to a JDBC BIT type

  Registered under `boolean` and `java.lang.Boolean` in the type registry (see *Seção 6.5, "Type registry"*).

`org.hibernate.type.NumericBooleanType`
: Maps a boolean to a JDBC INTEGER type as 0 = false, 1 = true

  Registered under `numeric_boolean` in the type registry (see *Seção 6.5, "Type registry"*).

`org.hibernate.type.YesNoType`
: Maps a boolean to a JDBC CHAR type as ('N' | 'n') = false, ('Y' | 'y') = true

  Registered under `yes_no` in the type registry (see *Seção 6.5, "Type registry"*).

`org.hibernate.type.TrueFalseType`
: Maps a boolean to a JDBC CHAR type as ('F' | 'f') = false, ('T' | 't') = true

  Registered under `true_false` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.4. `java.lang.Byte` (or byte primitive)

`org.hibernate.type.ByteType`
: Maps a byte or `java.lang.Byte` to a JDBC TINYINT

  Registered under `byte` and `java.lang.Byte` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.5. `java.lang.Short` (or short primitive)

`org.hibernate.type.ShortType`
Maps a short or `java.lang.Short` to a JDBC SMALLINT

Registered under `short` and `java.lang.Short` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.6. `java.lang.Integer` (or int primitive)

`org.hibernate.type.IntegerTypes`
Maps an int or `java.lang.Integer` to a JDBC INTEGER

Registered under `int` and `java.lang.Integer`in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.7. `java.lang.Long` (or long primitive)

`org.hibernate.type.LongType`
Maps a long or `java.lang.Long` to a JDBC BIGINT

Registered under `long` and `java.lang.Long` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.8. `java.lang.Float` (or float primitive)

`org.hibernate.type.FloatType`
Maps a float or `java.lang.Float` to a JDBC FLOAT

Registered under `float` and `java.lang.Float` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.9. `java.lang.Double` (or double primitive)

`org.hibernate.type.DoubleType`
Maps a double or `java.lang.Double` to a JDBC DOUBLE

Registered under `double` and `java.lang.Double` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.10. `java.math.BigInteger`

`org.hibernate.type.BigIntegerType`
Maps a `java.math.BigInteger` to a JDBC NUMERIC

Registered under `big_integer` and `java.math.BigInteger` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.11. `java.math.BigDecimal`

`org.hibernate.type.BigDecimalType`
Maps a `java.math.BigDecimal` to a JDBC NUMERIC

Registered under `big_decimal` and `java.math.BigDecimal` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.12. `java.util.Date` or `java.sql.Timestamp`

`org.hibernate.type.TimestampType`
Maps a `java.sql.Timestamp` to a JDBC TIMESTAMP

Registered under `timestamp`, `java.sql.Timestamp` and `java.util.Date` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.13. `java.sql.Time`

`org.hibernate.type.TimeType`
Maps a `java.sql.Time` to a JDBC TIME

Registered under `time` and `java.sql.Time` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.14. `java.sql.Date`

`org.hibernate.type.DateType`
Maps a `java.sql.Date` to a JDBC DATE

Registered under `date` and `java.sql.Date` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.15. `java.util.Calendar`

`org.hibernate.type.CalendarType`
Maps a `java.util.Calendar` to a JDBC TIMESTAMP

Registered under `calendar`, `java.util.Calendar` and `java.util.GregorianCalendar` in the type registry (see *Seção 6.5, "Type registry"*).

`org.hibernate.type.CalendarDateType`
Maps a `java.util.Calendar` to a JDBC DATE

Registered under `calendar_date` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.16. `java.util.Currency`

`org.hibernate.type.CurrencyType`
Maps a `java.util.Currency` to a JDBC VARCHAR (using the Currency code)

Registered under `currency` and `java.util.Currency` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.17. `java.util.Locale`

`org.hibernate.type.LocaleType`
Maps a `java.util.Locale` to a JDBC VARCHAR (using the Locale code)

Registered under `locale` and `java.util.Locale` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.18. `java.util.TimeZone`

`org.hibernate.type.TimeZoneType`
Maps a `java.util.TimeZone` to a JDBC VARCHAR (using the TimeZone ID)

Registered under `timezone` and `java.util.TimeZone` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.19. `java.net.URL`

`org.hibernate.type.UrlType`
Maps a `java.net.URL` to a JDBC VARCHAR (using the external form)

Registered under `url` and `java.net.URL` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.20. `java.lang.Class`

`org.hibernate.type.ClassType`
Maps a `java.lang.Class` to a JDBC VARCHAR (using the Class name)

Registered under `class` and `java.lang.Class` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.21. `java.sql.Blob`

`org.hibernate.type.BlobType`
Maps a `java.sql.Blob` to a JDBC BLOB

Registered under `blob` and `java.sql.Blob` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.22. `java.sql.Clob`

`org.hibernate.type.ClobType`
Maps a `java.sql.Clob` to a JDBC CLOB

Registered under `clob` and `java.sql.Clob` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.23. byte[]

`org.hibernate.type.BinaryType`
Maps a primitive byte[] to a JDBC VARBINARY

Registered under `binary` and `byte[]` in the type registry (see *Seção 6.5, "Type registry"*).

`org.hibernate.type.MaterializedBlobType`
Maps a primitive byte[] to a JDBC BLOB

Registered under `materialized_blob` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.24. byte[]

`org.hibernate.type.BinaryType`
Maps a java.lang.Byte[] to a JDBC VARBINARY

Registered under `wrapper-binary`, `Byte[]` and `java.lang.Byte[]` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.25. char[]

`org.hibernate.type.CharArrayType`
Maps a char[] to a JDBC VARCHAR

Registered under `characters` and `char[]` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.26. char[]

`org.hibernate.type.CharacterArrayType`
Maps a java.lang.Character[] to a JDBC VARCHAR

Registered under `wrapper-characters`, `Character[]` and `java.lang.Character[]` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.27. `java.util.UUID`

`org.hibernate.type.UUIDBinaryType`
Maps a java.util.UUID to a JDBC BINARY

Registered under `uuid-binary` and `java.util.UUID` in the type registry (see *Seção 6.5, "Type registry"*).

`org.hibernate.type.UUIDCharType`
Maps a java.util.UUID to a JDBC CHAR (though VARCHAR is fine too for existing schemas)

Registered under `uuid-char` in the type registry (see *Seção 6.5, "Type registry"*).

`org.hibernate.type.PostgresUUIDType`
: Maps a java.util.UUID to the PostgreSQL UUID data type (through `Types#OTHER` which is how the PostgreSQL JDBC driver defines it).

  Registered under `pg-uuid` in the type registry (see *Seção 6.5, "Type registry"*).

6.1.1.28. `java.io.Serializable`

`org.hibernate.type.SerializableType`
: Maps implementors of java.lang.Serializable to a JDBC VARBINARY

  Unlike the other value types, there are multiple instances of this type. It gets registered once under `java.io.Serializable`. Additionally it gets registered under the specific `java.io.Serializable` implementation class names.

6.1.2. Composite types

Nota

The Java Persistence API calls these embedded types, while Hibernate traditionally called them components. Just be aware that both terms are used and mean the same thing in the scope of discussing Hibernate.

Components represent aggregations of values into a single Java type. For example, you might have an Address class that aggregates street, city, state, etc information or a Name class that aggregates the parts of a person's Name. In many ways a component looks exactly like an entity. They are both (generally speaking) classes written specifically for the application. They both might have references to other application-specific classes, as well as to collections and simple JDK types. As discussed before, the only distinguishing factory is the fact that a component does not own its own lifecycle nor does it define an identifier.

6.1.3. Collection types

Importante

It is critical understand that we mean the collection itself, not its contents. The contents of the collection can in turn be basic, component or entity types (though not collections), but the collection itself is owned.

Collections are covered in *Capítulo 7, Mapeamento de coleção* .

6.2. Entity types

The definition of entities is covered in detail in *Capítulo 4, Classes Persistentes*. For the purpose of this discussion, it is enough to say that entities are (generally application-specific) classes which

correlate to rows in a table. Specifically they correlate to the row by means of a unique identifier. Because of this unique identifier, entities exist independently and define their own lifecycle. As an example, when we delete a `Membership`, both the `User` and `Group` entities remain.

Nota

This notion of entity independence can be modified by the application developer using the concept of cascades. Cascades allow certain operations to continue (or "cascade") across an association from one entity to another. Cascades are covered in detail in *Capítulo 8, Mapeamento de associações* .

6.3. Significance of type categories

Why do we spend so much time categorizing the various types of types? What is the significance of the distinction?

The main categorization was between entity types and value types. To review we said that entities, by nature of their unique identifier, exist independently of other objects whereas values do not. An application cannot "delete" a Product sku; instead, the sku is removed when the Product itself is deleted (obviously you can *update* the sku of that Product to null to make it "go away", but even there the access is done through the Product).

Nor can you define an association *to* that Product sku. You *can* define an association to Product *based on* its sku, assuming sku is unique, but that is totally different.

TBC...

6.4. Custom types

Hibernate makes it relatively easy for developers to create their own *value* types. For example, you might want to persist properties of type `java.lang.BigInteger` to `VARCHAR` columns. Custom types are not limited to mapping values to a single table column. So, for example, you might want to concatenate together `FIRST_NAME`, `INITIAL` and `SURNAME` columns into a `java.lang.String`.

There are 3 approaches to developing a custom Hibernate type. As a means of illustrating the different approaches, lets consider a use case where we need to compose a `java.math.BigDecimal` and `java.util.Currency` together into a custom `Money` class.

6.4.1. Custom types using `org.hibernate.type.Type`

The first approach is to directly implement the `org.hibernate.type.Type` interface (or one of its derivatives). Probably, you will be more interested in the more specific `org.hibernate.type.BasicType` contract which would allow registration of the type (see *Seção 6.5, “Type registry”*). The benefit of this registration is that whenever the metadata for a particular property does not specify the Hibernate type to use, Hibernate will consult the registry

for the exposed property type. In our example, the property type would be `Money`, which is the key we would use to register our type in the registry:

Exemplo 6.1. Defining and registering the custom Type

```
public class MoneyType implements BasicType {
    public String[] getRegistrationKeys() {
        return new String[] { Money.class.getName() };
    }

        public int[] sqlTypes(Mapping mapping) {
             // We will simply use delegation to the standard basic types for BigDecimal and
 Currency for many of the
            // Type methods...
            return new int[] {
                    BigDecimalType.INSTANCE.sqlType(),
                    CurrencyType.INSTANCE.sqlType(),
            };
            // we could also have honored any registry overrides via...
            //return new int[] {
                                                    //
 mappings.getTypeResolver().basic( BigDecimal.class.getName() ).sqlTypes( mappings )[0],
                                                    //
 mappings.getTypeResolver().basic( Currency.class.getName() ).sqlTypes( mappings )[0]
            //};
        }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws SQLEx
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, boolean[] settable, SessionImplementor
            throws SQLException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}

Configuration cfg = new Configuration();
```

```
cfg.registerTypeOverride( new MoneyType() );
cfg...;
```

Importante

It is important that we registered the type *before* adding mappings.

6.4.2. Custom types using `org.hibernate.usertype.UserType`

Nota

Both `org.hibernate.usertype.UserType` and `org.hibernate.usertype.CompositeUserType` were originally added to isolate user code from internal changes to the `org.hibernate.type.Type` interfaces.

The second approach is the use the `org.hibernate.usertype.UserType` interface, which presents a somewhat simplified view of the `org.hibernate.type.Type` interface. Using a `org.hibernate.usertype.UserType`, our `Money` custom type would look as follows:

Exemplo 6.2. Defining the custom UserType

```
public class MoneyType implements UserType {
    public int[] sqlTypes() {
        return new int[] {
                BigDecimalType.INSTANCE.sqlType(),
                CurrencyType.INSTANCE.sqlType(),
        };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, Object owner) throws SQLException {
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

   public void nullSafeSet(PreparedStatement st, Object value, int index) throws SQLException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
```

```
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}
```

There is not much difference between the `org.hibernate.type.Type` example and the `org.hibernate.usertype.UserType` example, but that is only because of the snippets shown. If you choose the `org.hibernate.type.Type` approach there are quite a few more methods you would need to implement as compared to the `org.hibernate.usertype.UserType`.

6.4.3. Custom types using `org.hibernate.usertype.CompositeUserType`

The third and final approach is the use the `org.hibernate.usertype.CompositeUserType` interface, which differs from `org.hibernate.usertype.UserType` in that it gives us the ability to provide Hibernate the information to handle the composition within the `Money` class (specifically the 2 attributes). This would give us the capability, for example, to reference the `amount` attribute in an HQL query. Using a `org.hibernate.usertype.CompositeUserType`, our `Money` custom type would look as follows:

Exemplo 6.3. Defining the custom CompositeUserType

```
public class MoneyType implements CompositeUserType {
    public String[] getPropertyNames() {
          // ORDER IS IMPORTANT!   it must match the order the columns are defined in the
 property mapping
        return new String[] { "amount", "currency" };
    }

    public Type[] getPropertyTypes() {
        return new Type[] { BigDecimalType.INSTANCE, CurrencyType.INSTANCE };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object getPropertyValue(Object component, int propertyIndex) {
        if ( component == null ) {
            return null;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                return money.getAmount();
            }
            case 1: {
                return money.getCurrency();
            }
            default: {
```

```
                throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
    }

        public void setPropertyValue(Object component, int propertyIndex, Object value) throws HibernateException
        if ( component == null ) {
            return;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                money.setAmount( (BigDecimal) value );
                break;
            }
            case 1: {
                money.setCurrency( (Currency) value );
                break;
            }
            default: {
               throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
        }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws SQLEx
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, SessionImplementor session) throws SQLE
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}
```

6.5. Type registry

Internally Hibernate uses a registry of basic types (see *Seção 6.1.1, "Basic value types"*) when it needs to resolve the specific `org.hibernate.type.Type` to use in certain situations. It also provides a way for applications to add extra basic type registrations as well as override the standard basic type registrations.

To register a new type or to override an existing type registration, applications would make use of the `registerTypeOverride` method of the `org.hibernate.cfg.Configuration` class when bootstrapping Hibernate. For example, lets say you want Hibernate to use your custom `SuperDuperStringType`; during bootstrap you would call:

Exemplo 6.4. Overriding the standard `StringType`

```
Configuration cfg = ...;
cfg.registerTypeOverride( new SuperDuperStringType() );
```

The argument to `registerTypeOverride` is a `org.hibernate.type.BasicType` which is a specialization of the `org.hibernate.type.Type` we saw before. It adds a single method:

Exemplo 6.5. Snippet from BasicType.java

```
/**
         * Get the names under which this type should be registered in the type registry.
         *
         * @return The keys under which to register this type.
         */
        public String[] getRegistrationKeys();
```

One approach is to use inheritance (`SuperDuperStringType` extends `org.hibernate.type.StringType`); another is to use delegation.

Mapeamento de coleção

7.1. Coleções persistentes

Naturally Hibernate also allows to persist collections. These persistent collections can contain almost any other Hibernate type, including: basic types, custom types, components and references to other entities. The distinction between value and reference semantics is in this context very important. An object in a collection might be handled with "value" semantics (its life cycle fully depends on the collection owner), or it might be a reference to another entity with its own life cycle. In the latter case, only the "link" between the two objects is considered to be a state held by the collection.

As a requirement persistent collection-valued fields must be declared as an interface type (see *Exemplo 7.2, "Collection mapping using @OneToMany and @JoinColumn"*). The actual interface might be `java.util.Set`, `java.util.Collection`, `java.util.List`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap` or anything you like ("anything you like" means you will have to write an implementation of `org.hibernate.usertype.UserCollectionType`).

Notice how in *Exemplo 7.2, "Collection mapping using @OneToMany and @JoinColumn"* the instance variable `parts` was initialized with an instance of `HashSet`. This is the best way to initialize collection valued properties of newly instantiated (non-persistent) instances. When you make the instance persistent, by calling `persist()`, Hibernate will actually replace the `HashSet` with an instance of Hibernate's own implementation of `Set`. Be aware of the following error:

Exemplo 7.1. Hibernate uses its own collection implementations

```
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.

As instâncias de coleção têm o comportamento comum de tipos de valores. Eles são automaticamente persistidos quando referenciados por um objeto persistente e automaticamente deletados quando não referenciados. Se a coleção é passada de um objeto persistente para outro, seus elementos devem ser movidos de uma tabela para outra. Duas entidades não devem compartilhar uma referência com uma mesma instância de coleção. Devido ao modelo relacional

adjacente, as propriedades de coleções válidas, não suportam semânticas de valores nulos. O Hibernate não distingue entre a referência da coleção nula e uma coleção vazia.

Nota

Use persistent collections the same way you use ordinary Java collections. However, ensure you understand the semantics of bidirectional associations (see *Seção 7.3.2, "Associações Bidirecionais"*).

7.2. How to map collections

Using annotations you can map `Collections`, `Lists`, `Maps` and `Sets` of associated entities using @OneToMany and @ManyToMany. For collections of a basic or embeddable type use @ElementCollection. In the simplest case a collection mapping looks like this:

Exemplo 7.2. Collection mapping using @OneToMany and @JoinColumn

```
@Entity
public class Product {

    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();

    @Id
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinColumn(name="PART_ID")
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}


@Entity
public class Part {
   ...
}
```

Product describes a unidirectional relationship with Part using the join column PART_ID. In this unidirectional one to many scenario you can also use a join table as seen in *Exemplo 7.3, "Collection mapping using @OneToMany and @JoinTable"*.

Exemplo 7.3. Collection mapping using @OneToMany and @JoinTable

```
@Entity
public class Product {
```

```
    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();

    @Id
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinTable(
            name="PRODUCT_PARTS",
            joinColumns = @JoinColumn( name="PRODUCT_ID"),
            inverseJoinColumns = @JoinColumn( name="PART_ID")
    )
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}


@Entity
public class Part {
   ...
}
```

Without describing any physical mapping (no `@JoinColumn` or `@JoinTable`), a unidirectional one to many with join table is used. The table name is the concatenation of the owner table name, _, and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table, _, and the owner primary key column(s) name. The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s) name. A unique constraint is added to the foreign key referencing the other side table to reflect the one to many.

Lets have a look now how collections are mapped using Hibernate mapping files. In this case the first step is to chose the right mapping element. It depends on the type of interface. For example, a `<set>` element is used for mapping properties of type `Set`.

Exemplo 7.4. Mapping a Set using <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>
```

In *Exemplo 7.4, “Mapping a Set using <set>”* a *one-to-many association* links the `Product` and `Part` entities. This association requires the existence of a foreign key column and possibly an index column to the `Part` table. This mapping loses certain semantics of normal Java collections:

- Uma instância de classes entidades contidas, podem não pertencer à mais de uma instância da coleção.

- Uma instância da classe de entidade contida pode não aparecer em mais de um valor do índice da coleção.

Looking closer at the used `<one-to-many>` tag we see that it has the following options.

Exemplo 7.5. options of <one-to-many> element

```
<one-to-many
        class="ClassName"                          (1)
        not-found="ignore|exception"               (2)
        entity-name="EntityName"                   (3)
        node="element-name"
        embed-xml="true|false"
    />
```

(1) `class` (requerido): O nome da classe associada.

(2) `not-found` (opcional - padrão para `exception`): Especifica como os identificadores em cache que referenciam as linhas faltantes serão tratadas: `ignore` tratará a linha faltante como uma associação nula.

(3) `entity-name` (opcional): O nome da entidade da classe associada, como uma alternativa para a `class`.

Note que o elemento `<one-to-many>` não precisa declarar qualquer coluna. Nem é necessário especificar o nome da `table` em qualquer lugar.

Atenção

If the foreign key column of a `<one-to-many>` association is declared `NOT NULL`, you must declare the `<key>` mapping `not-null="true"` or *use a bidirectional association* with the collection mapping marked `inverse="true"`. See *Seção 7.3.2, "Associações Bidirecionais"*.

Apart from the `<set>` tag as shown in *Exemplo 7.4, "Mapping a Set using <set>"*, there is also `<list>`, `<map>`, `<bag>`, `<array>` and `<primitive-array>` mapping elements. The `<map>` element is representative:

Exemplo 7.6. Elements of the <map> mapping

```
<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|delete-orphan"   6
    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 da coleção
2. `table` (opcional - padrão para nome de propriedade): o nome da tabela de coleção. Isto não é usado para associações um-para-muitos.
3. `schema` (opcional): o nome de um esquema de tabela para sobrescrever o esquema declarado no elemento raíz.
4. `lazy` (opcional - padrão para `true`): pode ser utilizado para desabilitar a busca lazy e especificar que a associação é sempre buscada antecipadamente, ou para habilitar busca "extra-lazy" onde a maioria das operações não inicializa a coleção (apropriado para coleções bem grandes).
5. `inverse` (opcional - padrão para `false`): marque esta coleção como o lado "inverso" de uma associação bidirecional.
6. `cascade` (opcional - padrão para `none`): habilita operações para cascata para entidades filho.
7. `sort` (opcional): especifica uma coleção escolhida com ordem de escolha`natural` ou uma dada classe comparatória.
8. `order-by` (optional): specifies a table column or columns that define the iteration order of the `Map`, `Set` or bag, together with an optional `asc` or `desc`.
9. `where` (opcional): especifica uma condição SQL arbitrária `WHERE` a ser usada quando recuperar ou remover a coleção Isto é útil se a coleção tiver somente um subconjunto dos dados disponíveis.
10. `fetch` (opcional, padrão para `select`): escolha entre busca de união externa, busca por seleção sequencial e busca por subseleção sequencial.
11. `batch-size` (opcional, padrão para `1`): especifica um "tamanho de lote" para instâncias de busca lazy desta coleção.
12. `access` (opcional - padrão para `property`): A estratégia que o Hibernate deve usar para acessar a coleção de valor de propriedade.

13. `optimistic-lock` (opcional - padrão para `true`): especifica que alterações para o estado da coleção, resulta no incremento da versão da própria entidade. Para associações um-para-muitos, é sempre bom desabilitar esta configuração.
14. `mutable` (opcional - padrão para `true`): um valor de `false` especifica que os elementos da coleção nunca mudam. Isto permite uma otimização mínima do desempenho em alguns casos.

After exploring the basic mapping of collections in the preceding paragraphs we will now focus details like physical mapping considerations, indexed collections and collections of value types.

7.2.1. Chaves Externas de Coleção

On the database level collection instances are distinguished by the foreign key of the entity that owns the collection. This foreign key is referred to as the *collection key column*, or columns, of the collection table. The collection key column is mapped by the `@JoinColumn` annotation respectively the `<key>` XML element.

There can be a nullability constraint on the foreign key column. For most collections, this is implied. For unidirectional one-to-many associations, the foreign key column is nullable by default, so you may need to specify

```
@JoinColumn(nullable=false)
```

or

```
<key column="productSerialNumber" not-null="true"/>
```

The foreign key constraint can use `ON DELETE CASCADE`. In XML this can be expressed via:

```
<key column="productSerialNumber" on-delete="cascade"/>
```

In annotations the Hibernate specific annotation @OnDelete has to be used.

```
@OnDelete(action=OnDeleteAction.CASCADE)
```

See *Seção 5.1.11.3, "Key"* for more information about the `<key>` element.

7.2.2. Coleções indexadas

In the following paragraphs we have a closer at the indexed collections `List` and `Map` how the their index can be mapped in Hibernate.

7.2.2.1. Lists

Lists can be mapped in two different ways:

- as ordered lists, where the order is not materialized in the database
- as indexed lists, where the order is materialized in the database

To order lists in memory, add `@javax.persistence.OrderBy` to your property. This annotation takes as parameter a list of comma separated properties (of the target entity) and orders the collection accordingly (eg `firstname asc, age desc`), if the string is empty, the collection will be ordered by the primary key of the target entity.

Exemplo 7.7. Ordered lists using `@OrderBy`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderBy("number")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------|
| Order       | | Customer |
|-------------| |----------|
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

To store the index value in a dedicated column, use the `@javax.persistence.OrderColumn` annotation on your property. This annotations describes the column name and attributes of the column keeping the index value. This column is hosted on the table containing the association foreign key. If the column name is not specified, the default is the name of the referencing property, followed by underscore, followed by `ORDER` (in the following example, it would be `orders_ORDER`).

Exemplo 7.8. Explicit index column using `@OrderColumn`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderColumn(name="orders_index")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|--------------| |----------|
| Order        | | Customer |
|--------------| |----------|
| id           | | id       |
| number       | |----------|
| customer_id  |
| orders_order |
|--------------|
```

Nota

We recommend you to convert the legacy `@org.hibernate.annotations.IndexColumn` usages to `@OrderColumn` unless you are making use of the base property. The `base` property lets you define the

index value of the first element (aka as base index). The usual value is `0` or `1`. The default is 0 like in Java.

Looking again at the Hibernate mapping file equivalent, the index of an array or list is always of type `integer` and is mapped using the `<list-index>` element. The mapped column contains sequential integers that are numbered from zero by default.

Exemplo 7.9. index-list element for indexed collections in xml mapping

```
<list-index
        column="column_name"                                    (1)
        base="0|1|..."/>
```

(1) `column_name` (requerido): O nome da coluna que mantém os valores do índice da coleção.

(1) `base` (opcional - padrão para `0`): o valor da coluna de índice que corresponde ao primeiro elemento da lista ou matriz.

Se sua tabela não possui uma coluna de índice e você ainda quiser usar a `Lista` como tipo de propriedade, você deve mapeiar a propriedade como uma *<bag>* do Hibernate. Uma bag não mantém sua ordem quando é recuperadada do banco de dados, mas pode ser escolhida de forma opcional ou ordenada.

7.2.2.2. Maps

The question with `Maps` is where the key value is stored. There are everal options. Maps can borrow their keys from one of the associated entity properties or have dedicated columns to store an explicit key.

To use one of the target entity property as a key of the map, use `@MapKey(name="myProperty")`, where `myProperty` is a property name in the target entity. When using `@MapKey` without the name attribuate, the target entity primary key is used. The map key uses the same column as the property pointed out. There is no additional column defined to hold the map key, because the map key represent a target property. Be aware that once loaded, the key is no longer kept in sync with the property. In other words, if you change the property value, the key will not change automatically in your Java model.

Exemplo 7.10. Use of target entity property as map key via `@MapKey`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @MapKey(name="number")
```

```
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> order) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------|
| Order       | | Customer |
|-------------| |----------|
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

Alternatively the map key is mapped to a dedicated column or columns. In order to customize the mapping use one of the following annotations:

- `@MapKeyColumn` if the map key is a basic type. If you don't specify the column name, the name of the property followed by underscore followed by `KEY` is used (for example `orders_KEY`).
- `@MapKeyEnumerated` / `@MapKeyTemporal` if the map key type is respectively an enum or a `Date`.
- `@MapKeyJoinColumn`/`@MapKeyJoinColumns` if the map key type is another entity.
- `@AttributeOverride`/`@AttributeOverrides` when the map key is a embeddable object. Use `key.` as a prefix for your embeddable object property names.

You can also use `@MapKeyClass` to define the type of the key if you don't use generics.

Exemplo 7.11. Map key as basic type using `@MapKeyColumn`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany @JoinTable(name="Cust_Order")
   @MapKeyColumn(name="orders_number")
```

```
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> orders) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------| |---------------|
| Order       | | Customer | | Cust_Order    |
|-------------| |----------| |---------------|
| id          | | id       | | customer_id   |
| number      | |----------| | order_id      |
| customer_id |              | orders_number |
|-------------|              |---------------|
```

Nota

We recommend you to migrate from `@org.hibernate.annotations.MapKey` / `@org.hibernate.annotation.MapKeyManyToMany` to the new standard approach described above

Using Hibernate mapping files there exists equivalent concepts to the descibed annotations. You have to use `<map-key>`, `<map-key-many-to-many>` and `<composite-map-key>`. `<map-key>` is used for any basic type, `<map-key-many-to-many>` for an entity reference and `<composite-map-key>` for a composite type.

Exemplo 7.12. map-key xml mapping element

```
<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 mnatém os valores do índice de coleção.
2. `formula` (opcional): Uma fórmula SQL usada para avaliar a chave ou o mapa.
3. `type` (requerido): o tipo de chaves de mapa.

Exemplo 7.13. map-key-many-to-many

```
<map-key-many-to-many
        column="column_name"                    (1)
        formula="any SQL expression"            (2)(3)
        class="ClassName"
/>
```

1. `column` (opcional): o nome de uma coluna de chave exterior para os valores do índice de coleção.
2. `formula` (opcional): uma fórmula SQ usada para avaliar a chave exterior da chave do mapa.
3. `class` (requerido): a classe da entidade usada como chave do mapa.

7.2.3. Collections of basic types and embeddable objects

In some situations you don't need to associate two entities but simply create a collection of basic types or embeddable objects. Use the `@ElementCollection` for this case.

Exemplo 7.14. Collection of basic types mapped via `@ElementCollection`

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Nicknames", joinColumns=@JoinColumn(name="user_id"))
   @Column(name="nickname")
   public Set<String> getNicknames() { ... }
}
```

The collection table holding the collection data is set using the `@CollectionTable` annotation. If omitted the collection table name defaults to the concatenation of the name of the containing entity and the name of the collection attribute, separated by an underscore. In our example, it would be `User_nicknames`.

The column holding the basic type is set using the `@Column` annotation. If omitted, the column name defaults to the property name: in our example, it would be `nicknames`.

But you are not limited to basic types, the collection type can be any embeddable object. To override the columns of the embeddable object in the collection table, use the `@AttributeOverride` annotation.

Exemplo 7.15. @ElementCollection for embeddable objects

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Addresses", joinColumns=@JoinColumn(name="user_id"))
   @AttributeOverrides({
      @AttributeOverride(name="street1", column=@Column(name="fld_street"))
   })
   public Set<Address> getAddresses() { ... }
}

@Embeddable
public class Address {
   public String getStreet1() {...}
   [...]
}
```

Such an embeddable object cannot contains a collection itself.

Nota

in `@AttributeOverride`, you must use the `value.` prefix to override properties of the embeddable object used in the map value and the `key.` prefix to override properties of the embeddable object used in the map key.

```
@Entity
public class User {
   @ElementCollection
   @AttributeOverrides({
      @AttributeOverride(name="key.street1", column=@Column(name="fld_street")),
      @AttributeOverride(name="value.stars", column=@Column(name="fld_note"))
   })
   public Map<Address,Rating> getFavHomes() { ... }
```

Nota

We recommend you to migrate from `@org.hibernate.annotations.CollectionOfElements` to the new `@ElementCollection` annotation.

Using the mapping file approach a collection of values is mapped using the `<element>` tag. For example:

Exemplo 7.16. <element> tag for collection values using mapping files

```
<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 mantém os valores do elemento da coleção.

(2) `formula` (opcional): uma fórmula usada para avaliar o elemento.

(3) `type` (requerido): o tipo do elemento de coleção.

7.3. Mapeamentos de coleção avançados.

7.3.1. Coleções escolhidas

Hibernate supports collections implementing `java.util.SortedMap` and `java.util.SortedSet`. With annotations you declare a sort comparator using `@Sort`. You chose between the comparator types unsorted, natural or custom. If you want to use your own comparator implementation, you'll also have to specify the implementation class using the `comparator` attribute. Note that you need to use either a `SortedSet` or a `SortedMap` interface.

Exemplo 7.17. Sorted collection with @Sort

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Sort(type = SortType.COMPARATOR, comparator = TicketComparator.class)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

Using Hibernate mapping files you specify a comparator in the mapping file with `<sort>`:

Exemplo 7.18. Sorted collection using xml mapping

```
<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>
```

Valores permitidos da função`sort` sãounsorted, `natural` e o nome de uma classe implementando `java.util.Comparator`.

Dica

Coleções escolhidas, na verdade se comportam como `java.util.TreeSet` ou `java.util.TreeMap`.

If you want the database itself to order the collection elements, use the `order-by` attribute of `set`, `bag` or `map` mappings. This solution is implemented using `LinkedHashSet` or `LinkedHashMap` and performs the ordering in the SQL query and not in the memory.

Exemplo 7.19. Sorting in database using order-by

```
<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>
```

Nota

Note que o valor da função `order-by` é uma ordenação SQL e não uma ordenação.

Associações podem também ser escolhidas por algum critério arbritrário em tempo de espera usando uma coleção `filter()`:

Exemplo 7.20. Sorting via a query filter

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.name" ).list();
```

7.3.2. Associações Bidirecionais

Uma *associação bidirecional* permite a navegação de ambos os "lados" da associação. Dois dos casos de associação bidirecional, são suportados:

Um-para-muitos
: conjunto ou bag de valor em um dos lados, valor único do outro

Muitos-para-muitos
: Conjunto ou bag com valor em ambos os lados

Often there exists a many to one association which is the owner side of a bidirectional relationship. The corresponding one to many association is in this case annotated by `@OneToMany(mappedBy=...)`

Exemplo 7.21. Bidirectional one to many with many to one side as association owner

```
@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
    ...
}
```

`Troop` has a bidirectional one to many relationship with `Soldier` through the `troop` property. You don't have to (must not) define any physical mapping in the `mappedBy` side.

To map a bidirectional one to many, with the one-to-many side as the owning side, you have to remove the `mappedBy` element and set the many to one `@JoinColumn` as insertable and updatable to false. This solution is not optimized and will produce additional UPDATE statements.

Exemplo 7.22. Bidirectional associtaion with one to many side as owner

```
@Entity
```

```
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") //we need to duplicate the physical information
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk", insertable=false, updatable=false)
    public Troop getTroop() {
    ...
}
```

How does the mappping of a bidirectional mapping look like in Hibernate mapping xml? There you define a bidirectional one-to-many association by mapping a one-to-many association to the same table column(s) as a many-to-one association and declaring the many-valued end `inverse="true"`.

Exemplo 7.23. Bidirectional one to many via Hibernate mapping files

```
<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, uma vez que isto é um conceito ortogonal.

A many-to-many association is defined logically using the `@ManyToMany` annotation. You also have to describe the association table and the join conditions using the `@JoinTable` annotation. If the association is bidirectional, one side has to be the owner and one side has to be the inverse end (ie. it will be ignored when updating the relationship values in the association table):

Exemplo 7.24. Many to many association via @ManyToMany

```
@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns=@JoinColumn(name="EMPER_ID"),
        inverseJoinColumns=@JoinColumn(name="EMPEE_ID")
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}
```

```
@Entity
public class Employee implements Serializable {
    @ManyToMany(
        cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        mappedBy = "employees",
        targetEntity = Employer.class
    )
    public Collection getEmployers() {
        return employers;
    }
}
```

In this example `@JoinTable` defines a `name`, an array of join columns, and an array of inverse join columns. The latter ones are the columns of the association table which refer to the `Employee` primary key (the "other side"). As seen previously, the other side don't have to (must not) describe the physical mapping: a simple `mappedBy` argument containing the owner side property name bind the two.

As any other annotations, most values are guessed in a many to many relationship. Without describing any physical mapping in a unidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table name, _ and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

Exemplo 7.25. Default values for `@ManyToMany` (uni-directional)

```
@Entity
```

```
public class Store {
    @ManyToMany(cascade = CascadeType.PERSIST)
    public Set<City> getImplantedIn() {
        ...
    }
}

@Entity
public class City {
    ... //no bidirectional relationship
}
```

A `Store_City` is used as the join table. The `Store_id` column is a foreign key to the `Store` table. The `implantedIn_id` column is a foreign key to the `City` table.

Without describing any physical mapping in a bidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the other side property name, _, and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

Exemplo 7.26. Default values for `@ManyToMany` (bi-directional)

```
@Entity
public class Store {
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    public Set<Customer> getCustomers() {
        ...
    }
}

@Entity
public class Customer {
    @ManyToMany(mappedBy="customers")
    public Set<Store> getStores() {
        ...
    }
}
```

A `Store_Customer` is used as the join table. The `stores_id` column is a foreign key to the `Store` table. The `customers_id` column is a foreign key to the `Customer` table.

Using Hibernate mapping files you can map a bidirectional many-to-many association by mapping two many-to-many associations to the same database table and declaring one end as *inverse*.

Nota

You cannot select an indexed collection.

Exemplo 7.27, "Many to many association using Hibernate mapping files" shows a bidirectional many-to-many association that illustrates how each category can have many items and each item can be in many categories:

Exemplo 7.27. Many to many association using Hibernate mapping files

```
<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="ITEM_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 de objetos do Java e como criamos um relacionamento muitos para muitos em Java:

Exemplo 7.28. Effect of inverse vs. non-inverse side of many to many associations

```
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 para salvar a representação em memória à base de dados.

7.3.3. Associações bidirecionais com coleções indexadas

There are some additional considerations for bidirectional mappings with indexed collections (where one end is represented as a `<list>` or `<map>`) when using Hibernate mapping files. If there

is a property of the child class that maps to the index column you can use `inverse="true"` on the collection mapping:

Exemplo 7.29. Bidirectional association with indexed collection

```
<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, nós não podemos mapear a coleção usando `inverse="true"`. Devemos usar o seguinte mapeamento:

Exemplo 7.30. Bidirectional association with indexed collection, but no index column

```
<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, o lado de coleção válida da associação é responsável pela atualização da chave exterior.

7.3.4. Associações Ternárias

Há três meios possíveis de se mapear uma associação ternária. Uma é usar um `Map` com uma associação como seu índice:

Exemplo 7.31. Ternary association mapping

```
@Entity
public class Company {
   @Id
   int id;
   ...
   @OneToMany // unidirectional
   @MapKeyJoinColumn(name="employee_id")
   Map<Employee, Contract> contracts;
}

// or

<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>
```

A second approach is to remodel the association as an entity class. This is the most common approach. A final alternative is to use composite elements, which will be discussed later.

7.3.5. `Using an <idbag>`

The majority of the many-to-many associations and collections of values shown previously all map to tables with composite keys, even though it has been suggested that entities should have synthetic identifiers (surrogate keys). A pure association table does not seem to benefit much from a surrogate key, although a collection of composite values *might*. For this reason Hibernate provides a feature that allows you to map many-to-many associations and collections of values to a table with a surrogate key.

O elemento `<idbag>` permite mapear um `List` (ou uma `Collection`) com uma semântica de bag. Por exemplo:

```
<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>
```

O `<idbag>` possui um gerador de id sintético, igual a uma classe de entidade. Uma chave substituta diferente é associada para cada elemento de coleção. Porém, o Hibernate não provê de nenhum mecanismo para descobrir qual a chave substituta de uma linha em particular.

Note que o desempenho de atualização de um `<idbag>` é melhor do 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>`.

7.4. Exemplos de coleções

Esta sessão cobre os exemplos de coleções.

A seguinte classe possui uma coleção de instâncias `Child`:

Exemplo 7.32. Example classes `Parent` and `Child`

```
public class Parent {
    private long id;
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   private long id;
   private String name


   // getter/setter
   ...
}
```

Se cada Filho tiver no máximo um Pai, o mapeamento natural é uma associação um para muitos:

Exemplo 7.33. One to many unidirectional `Parent-Child` relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;
   private String name;


   // getter/setter
   ...
}
```

Exemplo 7.34. One to many unidirectional `Parent-Child` relationship using mapping files

```
<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

Exemplo 7.35. Table definitions for unidirectional `Parent-Child` relationship

```
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 pai for *obrigatório*, use uma associação bidirecional um para muitos:

Exemplo 7.36. One to many bidirectional `Parent-Child` relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(mappedBy="parent")
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;

   private String name;

   @ManyToOne
   private Parent parent;


   // getter/setter
   ...
}
```

Exemplo 7.37. One to many bidirectional `Parent-Child` relationship using mapping files

```
<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 restrição `NOT NULL`:

Exemplo 7.38. Table definitions for bidirectional `Parent-Child` relationship

```
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
```

Alternatively, if this association must be unidirectional you can enforce the `NOT NULL` constraint.

Exemplo 7.39. Enforcing NOT NULL constraint in unidirectional relation using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(optional=false)
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;
   private String name;


   // getter/setter
   ...
```

```
}
```

Exemplo 7.40. Enforcing NOT NULL constraint in unidirectional relation using mapping files

```
<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>
```

On the other hand, if a child has multiple parents, a many-to-many association is appropriate.

Exemplo 7.41. Many to many `Parent-Child` relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @ManyToMany
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;

   private String name;


   // getter/setter
```

```
   ...
}
```

Exemplo 7.42. Many to many `Parent-Child` relationship using mapping files

```
<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:

Exemplo 7.43. Table definitions for many to many releationship

```
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
```

For more examples and a complete explanation of a parent/child relationship mapping, see *Capítulo 24, Exemplo: Pai/Filho* for more information. Even more complex association mappings are covered in the next chapter.

Mapeamento de associações

8.1. Introdução

Os mapeamentos de associações são, geralmente, os mais difíceis de se acertar. Nesta seção nós examinaremos pelos casos canônicos um por um, começando com mapeamentos unidirecionais e considerando os casos bidirecionais. Usaremos `Person` e `Address` em todos os exemplos.

Classificaremos as associações pela sua multiplicidade e se elas mapeiam ou não uma intervenção na tabela associativa.

O uso de chaves externas anuláveis não é considerado uma boa prática na modelagem de dados tradicional, assim todos os nossos exemplos usam chaves externas anuláveis. Esta não é uma exigência do Hibernate, e todos os mapeamentos funcionarão se você remover as restrições de anulabilidade.

8.2. Associações Unidirecionais

8.2.1. 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 )
```

8.2.2. Um-para-um

Uma *associação unidirecional um-para-um em uma chave externa* é quase idêntica. A única diferença é a restrição única 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 chave primária* geralmente usa um gerador de id especial. 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 )
```

8.2.3. Um-para-muitos

Uma *associação unidirecional um-para-muitos em uma chave externa* 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 )
```

Acreditamos ser melhor usar uma tabela associativa para este tipo de associação.

8.3. Associações Unidirecionais com tabelas associativas

8.3.1. Um-para-muitos

Uma *associação um-para-muitos unidirecional usando uma tabela associativa* é o mais comum. Note que se especificarmos `unique="true"`, estaremos modificando a multiplicidade 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 )
```

8.3.2. 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 )
```

8.3.3. 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 )
```

8.3.4. 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
 (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

8.4. Associações Bidirecionais

8.4.1. Um-para-muitos/muitos-para-um

Uma *associação bidirecional muitos-para-um* é o tipo mais comum de associação. A seguinte amostra ilustra 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 precisará especificar a coluna `key` da chave externa como `not null`. O Hibernate administrará 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 ajustando `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
>
```

Caso uma coluna chave externa adjacente for `NOT NULL`, é importante que você defina `not-null="true"` no elemento `<key>` no mapeamento na coleção se a coluna de chave externa para `NOT NULL`. Não declare como `not-null="true"` apenas um elemento aninhado `<column>`, mas sim o elemento `<key>`.

8.4.2. Um-para-um

Uma *associação bidirecional um para um em uma chave externa* 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 )
```

8.5. Associações Bidirecionais com tabelas associativas

8.5.1. Um-para-muitos/muitos-para-um

Segue abaixo uma amostra da *associação bidirecional um para muitos em uma tabela de união*. Veja que `inverse="true"` pode ser colocado em qualquer ponta da associação, na coleção, ou na união.

```
<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 )
```

8.5.2. Um para um

Uma *associação bidirecional um-para-um em uma tabela de união* é algo bastante 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>
    <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 )
```

8.5.3. Muitos-para-muitos

Finalmente, nós temos uma *associação 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
 (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

8.6. Mapeamento de associações mais complexas

Uniões de associações mais complexas são *extremamente* raras. O Hibernate possibilita o tratamento de 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 as colunas `accountNumber`, `effectiveEndDate` e `effectiveStartDate`, mapeadas assim como segue:

```
<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 *atual*, aquela com `effectiveEndDate` nulo, 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 *mais recentemente* e empregado, aquele com a mais recente `startDate`, 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
>
```

Esta funcionalidade permite um grau de criatividade e flexibilidade, mas geralmente é mais prático tratar estes tipos de casos, usando uma pesquisa HQL ou uma pesquisa por critério.

Mapeamento de Componentes

A noção de *componente* é re-utilizada em vários contextos diferentes, para propósitos diferentes, pelo Hibernate.

9.1. 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 e 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 identificadoras.

Nosso mapeamento do Hibernate seria semelhante a este:

```
<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 person teria as seguintes colunas `pid`, `birthday`, `initial`, `first` and `last`.

Assim como todos tipos por valor, componentes não suportam referências cruzadas. Em outras palavras, duas persons podem ter o mesmo nome, mas os dois objetos person podem ter dois objetos de nome independentes, apenas "o mesmo" por valor. A semântica dos valores null de um componente são *ad hoc*. No recarregameno do conteúdo do objeto, o Hibernate entenderá 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(coleções, associações muitos-para-um, outros componentes, etc). Componentes agrupados *não* devem ser considerados luxo. O Hibernate tem a intenção de suportar um modelo de objetos fine-grained (muito bem granulados).

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.

```
<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
>
```

9.2. Coleções de objetos dependentes

Coleções de componentes são suportadas (ex.: uma matriz de tipo `Name`). Declare a 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
>
```

Importante

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 tiver componentes , use a tag `<nested-composite-element>`. Este é um caso bastante exótico – coleções de componentes que por si própria possui componentes. Neste momento você deve estar se perguntando se uma associação de um-para-muitos 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 são diferentes.

Um mapeamento de elemento composto não suporta propriedades capazes de serem null se você estiver usando um `<set>`. Não existe coluna chave primária separada na tabela de elemento composto. O Hibernate tem que usar cada valor das colunas para identificar um registro quando estiver deletando objetos, 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 que você mapeie 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
>
```

Não pode haver uma referência de compra 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 conjunto 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.

9.3. 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.

9.4. 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 um requisito 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 chave primária que depende da chave primária (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 chave exterior referenciando 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
>
```

Dica

O elemento `column` é uma alternativa para a função `column` em todos os lugares. O uso do elemento `column` apenas fornece mais opções de declaração, das quais são úteis quando utilizando `hbm2ddl`.

Uma associação `many-to-many` para `many-to-many` também usa a chave estrangeira 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 coleção de `OrderLine`s 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>` não declara colunas.

Se `OrderLine` possui uma coleção, ela também tem uma chave externa 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
>
```

9.5. 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 da implementação, 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 tempo de configuração do metamodelo do Hibernate através do objeto `Configuration`.

Mapeamento de Herança

10.1. As três estratégias

O Hibernate suporta as três estratégias básicas de mapeamento de herança:

- tabela por hierarquia de classes
- table per subclass
- tabela por classe concreta

Além disso, o Hibernate suporta um quarto tipo de polimorfismo um pouco diferente:

- polimorfismo implícito

É possível usar diferentes estratégias de mapeamento para diferentes ramificações da mesma hierarquia de herança. Você pode fazer uso do polimorfismo implícito para alcançá-lo através da hierarquia completa. De qualquer forma, o Hibernate não suporta a mistura de mapeamentos `<subclass>`, `<joined-subclass>` e `<union-subclass>` dentro do mesmo elemento raíz `<class>`. É possível usar, junto às estratégias, uma tabela por hierarquia e 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 que você estenda uma hierarquia de classes apenas adicionando um novo arquivo de mapeamento. Você deve especificar uma função `extends` no mapeamento da subclasse, nomeando uma superclasse previamente mapeada. 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
>
```

10.1.1. Tabela por hierarquia de classes

Vamos supor que temos uma interface `Payment`, com sua implementação `CreditCardPayment`, `CashPayment` e `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
>
```

É requisitado exatamente uma tabela. Existe uma grande limitação desta estratégia de mapeamento: colunas declaradas por subclasses, tais como `CCTYPE`, podem não ter restrições `NOT NULL`.

10.1.2. 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
>
```

São necessárias quatro tabelas. As três tabelas subclasses possuem associação de chave primária para a tabela de superclasse, desta maneira o modelo relacional é atualmente uma associação de um-para-um.

10.1.3. Tabela por subclasse: usando um discriminador

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 mais correto de um ponto de vista relacional. Se você deseja utilizar uma coluna discriminadora com a estratégia tabela por subclasse, você poderá 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`, quando usar uma união externa pesquisando a superclasse.

10.1.4. Mesclar tabela por hierarquia de classes com tabela por subclasse

Você pode até mesmo mesclar 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 raíz `Payment` deve ser mapeada usando `<many-to-one>`.

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

10.1.5. 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. A estratégia do gerador identidade não é permitida na união da herança de sub-classe. A fonte de chave primária deve ser compartilhada através de todas subclasses unidas da hierarquia.

Se sua superclasse é abstrata, mapeie-a com `abstract="true"`. Claro, que se ela não for abstrata, uma tabela adicional (padrão para `PAYMENT` no exemplo acima), será necessária para segurar as instâncias da superclasse.

10.1.6. 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. Note também que propriedades de `Payment` são mapeadas em cada uma das subclasses. Se você quiser evitar duplicação, considere usar entidades de XML (ex. `[ <!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 de 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
>
```

10.1.7. Mesclando polimorfismo implícito com outros mapeamentos de herança

Existe ainda um item a ser observado sobre este mapeamento. Como as subclasses são mapeadas em seu próprio elemento `<class>`, e como o `Payment` é 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`.

10.2. 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 seguinte tabela demonstra as limitações do mapeamento de tabela por classe concreta e do polimorfismo implícito no Hibernate.

Tabela 10.1. Recurso dos Mapeamentos de Herança

Estratégi de Herança	muitos-para-um Polimórfi	um-para-um Polimórfi	um-para-muitos Polimórfi	muitos-para-um Polimórfi	Polimórfi `load()/get()`	Consulta Polimórfi	Junções Polimórfi	Busca por união externa
tabela por hierarquia de class	`<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`	*supported*
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`	*supported*
tabela por classe concreta (subclasses de união)	`<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`	*supported*
tabela por classe concreta (polimorfismo implícito)	`<any>`	*not supported*	*not supported*	`<many-to-any>`	`s.createCriteria(Payment.class).add( Restrictions.`	`from Payment p`	*not supported*	*not supported*

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, como 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 considerar o *estado* de seus objetos, e não necessariamente a execução de instruções SQL. O Hibernate é responsável por esta parte e é relevante aos desenvolvedores de aplicações apenas quando estão ajustando o desempenho do sistema.

11.1. Estado dos objetos no Hibernate

O Hibernate define e suporta os seguintes estados de objetos:

- *Transient* - um objeto é transiente se ele foi instanciando usando apenas o operador `new` e não foi associado a uma `Session` do Hibernate. Ele não possui uma representação persistente no banco de dados e não lhe foi atribuído nenhum identificador. Instâncias transientes serão destruídas pelo coletor de lixo se a aplicação não mantiver 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. Ela pode ter sido salva ou carregada, portanto ela se encontra 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 se tornar transiente.
- *Detached* – uma instância desanexada é um objeto que foi persistido, mas sua `Session` foi fechada. A referência ao objeto continua válida, é claro, e a instância desanexada pode ser acoplada a uma nova `Session` no futuro, tornando-o novamente persistente (e todas as modificações sofridas). Essa característica habilita um modelo de programação para unidades de trabalho de longa execução, que requeira um tempo de espera do usuário. Podemos chamá-las de *transações da aplicação*, ou seja, 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.

11.2. Tornando os objetos persistentes

As instâncias recentemente instanciadas de uma classe persistente são consideradas *transientes* pelo Hibernate. Podemos transformar 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 à `cat` quando `save()` for chamado. Se `Cat` possuir um identificador `Associado`, ou uma chave composta, o identificador deverá ser atribuído à instância de `cat` antes que `save()` seja chamado. Pode-se usar também `persist()` ao invés de `save()`, com a semântica definida no novo esboço do EJB3.

- `persist()` faz uma instância transciente persistente. No entanto, isto não garante que o valor do identificador será determinado à instância persistente imediatamente, pois a determinação pode acontecer no período de limpeza. O `persist()` também garante que isto não executará uma declaração `INSERT` caso esta seja chamada fora dos limites da transação. Isto é útil em transações de longa-execução com um contexto de Sessão/persistência estendido.
- `save()` garante retornar um identificador. Caso um INSERT necessita ser executado para obter o identificador (ex.: gerador "identidade" e não "seqüência"), este INSERT acontece imediatamente, independente de você estar dentro ou fora da transação. Isto é problemático numa conversação de longa execução com um contexto de Sessão/persistência estendido.

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 tiver associado objetos (ex.: a coleção `kittens` no exemplo anterior), esses objetos podem se tornar persistentes em qualquer ordem que se queira, a não ser que se tenha uma restrição `NOT NULL` em uma coluna de chave estrangeira. Nunca há risco de violação de restrições de chave estrangeira. Assim, pode-se violar uma restrição `NOT NULL` se `save()` for usado nos objetos em uma ordem errada.

Geralmente você não precisa se preocupar com esses detalhes, pois muito provavelmente usará a característica de *persistência transitiva* do Hibernate para salvar os objetos associados automaticamente. Assim, enquanto uma restrição `NOT NULL` não ocorrer, o Hibernate tomará conta de tudo. Persistência transitiva será discutida mais adiante nesse mesmo capítulo.

11.3. Carregando o objeto

O método `load()` de uma `Session` oferece uma maneira de recuperar uma instância persistente se o identificador for conhecido. O `load()` escolhe uma classe do objeto e carregará o estado em uma instância mais recente dessa classe, em 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, pode-se 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 para 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 o `batch-size` estiver definido para o mapeamento da classe.

Se você não tiver certeza da existência do registro no banco, você deve usar o método `get()`, que consulta o banco imediatamente 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);
```

Note que quaisquer instâncias associadas ou que contenham coleções, *não* são selecionados `FOR UPDATE`, a não ser que você decida especificar um `lock` ou `all` como um estilo cascata para a associação.

É possível realizar o recarregamento de um objeto e todas as suas coleções a qualquer momento, usando o método `refresh()`.É útil quando os disparos 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)
```

How much does Hibernate load from the database and how many SQL `SELECT`s will it use? This depends on the *fetching strategy*. This is explained in *Seção 21.1, “Estratégias de Busca ”*.

11.4. Consultando

Se o identificador do objeto que se está buscando não for conhecido, será necessário realizar uma consulta. O Hibernate suporta uma linguagem de consulta (HQL) orientada a objetos fáceis de usar, porém poderosos. 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 resultados em objetos.

11.4.1. Executando consultas

Consultas HQL e SQL nativas 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 previamente, que a consulta retornará apenas um único objeto. Repare que consultas que fazem uso da busca antecipada (eager fetching) de coleções, geralmente retornam duplicatas dos objetos raiz, mas com suas coleções inicializadas. Pode-se filtrar estas duplicatas através de um simples `Set`.

11.4.1.1. Interagindo com resultados

Ocasionalmente, pode-se obter um melhor desempenho com a execução de consultas, usando o método `iterate()`. Geralmente isso acontece apenas se as instâncias das entidades reais retornadas pela consulta já estiverem na sessão ou no cachê de segundo nível. Caso elas ainda não tenham sido armazenadas, `iterate()` será mais devagar do que `list()` e podem ser necessários vários acessos ao banco de dados para uma 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;
    }
}
```

11.4.1.2. Consultas que retornam tuplas

Algumas vezes as consultas do Hibernate retornam tuplas de objetos. Cada tupla é retornada como uma matriz:

```
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 = (Cat) tuple[0];
    Cat mother = (Cat) tuple[1];
    ....
}
```

11.4.1.3. Resultados escalares

As consultas devem especificar uma propriedade da classe na cláusula `select`. Elas também podem chamar funções SQL de agregações. Propriedades ou agregações são consideradas 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];
    .....
}
```

11.4.1.4. Parâmetros de vínculo

Métodos em `Consulta` são oferecidos para valores de vínculo para parâmetros nomeados ou de estilo JDBC `?`. *Ao contrário do JDBC, o Hibernate numera parâmetros a partir de zero.* Parâmetros nomeados são identificadores da forma `:name` na faixa de consulta. As vantagens de parâmetros nomeados são:

- Parâmetros nomeados são insensíveis à ordem que eles ocorrem na faixa de consulta
- eles podem ocorrer em tempos múltiplos na mesma consulta
- eles são auto documentáveis

```
//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();
```

11.4.1.5. Paginação

Se você precisar especificar vínculos do conjunto de resultados, o máximo de números por linha que quiser recuperar e/ou a primeira linha que quiser recuperar, você deve usar métodos de interface `Consulta`:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

O Hibernate sabe como traduzir esta consulta de limite para a SQL nativa de seu DBMS

11.4.1.6. Iteração rolável

Se seu driver JDBC driver suportar `ResultSets` roláveis, a interface da `Consulta` poderá ser usada para obter um objeto de `ScrollableResults`, que permite uma navegação 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()
```

Note que uma conexão aberta de banco de dados (e cursor) é requerida para esta função, use `setMaxResult()`/`setFirstResult()` se precisar da função de paginação offline.

11.4.1.7. Externando consultas nomeadas

Queries can also be configured as so called named queries using annotations or Hibernate mapping documents. `@NamedQuery` and `@NamedQueries` can be defined at the class level as seen in *Exemplo 11.1, "Defining a named query using @NamedQuery"* . However their definitions are global to the session factory/entity manager factory scope. A named query is defined by its name and the actual query string.

Exemplo 11.1. Defining a named query using `@NamedQuery`

```
@Entity
@NamedQuery(name="night.moreRecentThan", query="select n from Night n where n.date >= :date")
public class Night {
    ...
}

public class MyDao {
    doStuff() {
        Query q = s.getNamedQuery("night.moreRecentThan");
        q.setDate( "date", aMonthAgo );
        List results = q.list();
        ...
    }
    ...
}
```

Using a mapping document can be configured using the `<query>` node. Remember to use a `CDATA` section if your query contains characters that could be interpreted as markup.

Exemplo 11.2. Defining a named query using `<query>`

```
<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

Parameter binding and executing is done programatically as seen in *Exemplo 11.3, "Parameter binding of a named query"*.

Exemplo 11.3. Parameter binding of a named query

```
Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

Note que o código de programa atual é independente da linguagem de consulta que é utilizada, você também pode definir as consultas SQL nativas no metadado, ou migrar consultas existentes para o Hibernate, colocando-os em arquivos de mapeamento.

Observe também que uma declaração de consulta dentro de um elemento `<hibernate-mapping>` requer um nome único global para a consulta, enquanto uma declaração de consulta dentro de um elemento de `<classe>` torna-se único automaticamente, aguardando o nome completo da classe qualificada, por exemplo: `eg.Cat.ByNameAndMaximumWeight`.

11.4.2. Filtrando coleções

Uma coleção *filter* é um tipo especial de consulta que pode ser aplicado a uma coleção persistente ou a uma matriz. A faixa de consulta pode referir-se ao `this`, significando o elemento de coleção atual.

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
);
```

A coleção retornada é considerada uma bolsa, e é a cópia da coleção dada. A coleção original não é modificada. Ela é oposta à implicação do nome "filtro", mas é consistente com o comportamento esperado.

Observe que os filtros não requerem uma cláusula `from` embora possam ter um, se requerido. Os filtros não são limitados a retornar aos elementos de coleção.

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue")
    .list();
```

Até mesmo um filtro vazio é útil, ex.: para carregar um subconjunto em uma coleção enorme:

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
```

```
    .setFirstResult(0).setMaxResults(10)
    .list();
```

11.4.3. Consulta por critério

O HQL é extremamente potente mas alguns desenvolvedores preferem construir consultas de forma dinâmica, utilizando um API de objeto, ao invés de construir faixas de consultas. O Hibernate oferece uma API de consulta de `Critério` intuitiva para estes casos:

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Restrictions.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

The `Criteria` and the associated `Example` API are discussed in more detail in *Capítulo 17, Consultas por critérios*.

11.4.4. Consultas em SQL nativa

Você pode expressar uma consulta em SQL utilizando `createSQLQuery()` e deixar o Hibernate tomar conta do mapeamento desde conjuntos de resultados até objetos. Note que você pode chamar uma `session.connection()` a qualquer momento e usar a `Connection` JDBC diretamente. Se você escolher utilizar a API Hibernate, você deve incluir as aliases SQL dentro de chaves:

```
List cats = session.createSQLQuery("SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10")
    .addEntity("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")
    .addEntity("cat", Cat.class)
.list()
```

SQL queries can contain named and positional parameters, just like Hibernate queries. More information about native SQL queries in Hibernate can be found in *Capítulo 18, SQL Nativo*.

11.5. Modificando objetos persistentes

Instâncias persistentes transacionais (ou seja, objetos carregados, salvos, criados ou consultados pela `Session`) podem ser manipuladas pela aplicação e qualquer mudança para estado persistente será persistida quando a `Sessão` for *limpa*. Isto será discutido mais adiante neste

capítulo. Não há necessidade de chamar um método em particular (como `update()`, que possui um propósito diferente) para fazer modificações persistentes. Portanto, a forma mais direta de atualizar o estado de um objeto é `carregá-lo()` e depois manipulá-lo diretamente, enquanto a `Sessão` 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
```

Algumas vezes, este modelo de programação é ineficiente, uma vez que ele requer ambos SQL `SELECT` (para carregar um objeto) e um SQL`UPDATE` (para persistir seu estado atualizado) na mesma sessão. Por isso, o Hibernate oferece uma abordagem alternativa, usando instâncias desanexadas.

11.6. Modificando objetos desacoplados

Muitas aplicações precisam recuperar um objeto em uma transação, enviá-lo para a camada UI para manipulação e somente então salvar as mudanças em uma nova transação. As aplicações que usam este tipo de abordagem em ambiente de alta concorrência, geralmente usam dados versionados para assegurar isolação durante a "longa" unidade de trabalho.

O Hibernate suporta este modelo, oferecendo re-acoplamentos das instâncias usando os métodos `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 identificador `catId` já tivesse sido carregado pela`segundaSessão` quando a aplicação tentou re-acoplá-lo, teria surgido uma exceção.

Use `update()` se você tiver certeza de que a sessão já não contém uma instância persistente com o mesmo identificador, e `merge()` se você quiser mesclar suas modificações a qualquer momento sem considerar o estado da sessão. Em outras palavras, `update()` é geralmente o primeiro método que você chama em uma nova sessão, assegurando que o re-acoplamento de suas instâncias seja a primeira operação executada.

The application should individually `update()` detached instances that are reachable from the given detached instance *only* if it wants their state to be updated. This can be automated using *transitive persistence*. See *Seção 11.11, "Persistência Transitiva"* for more information.

O método `lock()` também permite que um aplicativo re-associe um objeto com uma nova sessão. No entanto, a instância desanexada não pode ser modificada.

```
//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);
```

Note que `lock()` pode ser usado com diversos `LockModes`, veja a documentação API e o capítulo sobre manuseio de transações para maiores informações. Re-acoplamento não é o único caso de uso para `lock()`.

Other models for long units of work are discussed in *Seção 13.3, "Controle de concorrência otimista"*.

11.7. Detecção automática de estado

Os usuários de Hibernate solicitaram um método geral, tanto para salvar uma instância transiente, gerando um novo identificador, quanto para atualizar/ re-acoplar as instâncias desanexadas associadas ao 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 semântica do `saveOrUpdate()` parecem ser confusos para novos usuários. A princípio, enquanto você não tentar usar instâncias de uma sessão em outra nova sessão, não precisará utilizar `update()`, `saveOrUpdate()`, ou `merge()`. Algumas aplicações inteiras nunca precisarão utilizar estes métodos.

Geralmente, `update()` ou `saveOrUpdate()`são utilizados nos seguintes cenários:

- a aplicação carrega um objeto na primeira sessão

- o objeto é passado para a camada UI
- algumas modificações são feitas ao objeto
- o objeto é retornado à camada lógica de negócios
- a aplicação persiste estas modificações, chamando `update()` em uma segunda sessão.

`saveOrUpdate()` faz o seguinte:

- se o objeto já estiver persistente nesta sessão, não faça nada
- se outro objeto associado com a sessão possuir o mesmo identificador, jogue uma exceção
- se o objeto não tiver uma propriedade de identificador `salve-o()`
- se o identificador do objeto possuir o valor atribuído ao objeto recentemente instanciado, `salve-o()`
- se o objeto for versionado por um `<version>` ou `<timestamp>`, e o valor da propriedade da versão for o mesmo valor atribuído ao objeto recentemente instanciado, `salve()` o mesmo
- do contrário `atualize()` o objeto

e a `mesclagem()` é bastante diferente:

- se existir uma instância persistente com um mesmo identificador associado atualmente com a sessão, copie o estado do objeto dado para a instância persistente.
- se não existir uma instância persistente atualmente associada com a sessão, tente carregá-la a partir do banco de dados, ou crie uma nova instância persistente
- a instância persistente é retornada
- a instância dada não se torna associada com a sessão, ela permanece desanexada

11.8. Apagando objetos persistentes

A `Session.delete()` removerá um estado de objeto do banco de dados. É claro que seu aplicativo pode ainda reter uma referência à um objeto apagado. É melhor pensar em `delete()` como fazer uma instância persistente se tornar transiente.

```
sess.delete(cat);
```

Você poderá deletar objetos na ordem que desejar, sem risco de violação de restrição da chave estrangeira. É possível violar uma restrição `NOT NULL` em uma coluna de chave estrangeira, apagando objetos na ordem inversa, ex.: se apagar o pai, mas esquecer de apagar o filho.

11.9. Replicando objeto entre dois armazenamentos de dados diferentes.

Algumas vezes é útil poder tirar um gráfico de instâncias persistentes e fazê-los persistentes em um armazenamento de dados diferente, sem gerar novamente valores de identificador.

```
//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 o `replicate()` irá lidar com conflitos em linhas existentes no banco de dados:

- `ReplicationMode.IGNORE`: ignore o objeto quando houver uma linha de banco de dados existente com o mesmo identificador.
- `ReplicationMode.OVERWRITE`: subscreva uma linha de banco de dados existente com um mesmo identificador.
- `ReplicationMode.EXCEPTION`: jogue uma exceção se houver uma linha de banco de dados existente com o mesmo identificador.
- `ReplicationMode.LATEST_VERSION`: subscreva a linha se seu número de versão for anterior ao número de versão do objeto, caso contrário, ignore o objeto.

O caso de uso para este recurso inclui dados de reconciliação em instâncias de banco de dados diferentes, atualizando informações da configuração do sistema durante a atualização do produto, retornando mudanças realizadas durante transações não ACID entre outras funções.

11.10. Limpando a Sessão

De vez em quando, a `Session` irá executar as instruções SQL, necessárias para sincronizar o estado de conexão JDBC com o estado de objetos mantidos na memória. Este processo de *flush*, ocorre por padrão nos seguintes pontos:

- antes de algumas execuções de consultas
- a partir de`org.hibernate.Transaction.commit()`
- a partir de `Session.flush()`

As instruções SQL são editadas na seguinte ordem:

1. todas as inserções de entidade, na mesma ordem que os objetos correspondentes foram salvos usando `Session.save()`
2. todas as atualizações de entidades

3. todas as deleções de coleções
4. todas as deleções, atualizações e inserções de elementos de coleção.
5. todas as inserções de coleção
6. todas as deleções de entidade, na mesma ordem que os objetos correspondentes foram deletados usando `Session.delete()`

Uma exceção é que o objeto que utiliza a geração de ID `native` é inserido quando salvo.

Exceto quando você explicitamente `limpar()`, não há nenhuma garantia sobre *quando* a `Sessão` executará as chamadas de JDBC, somente se sabe a *ordem* na qual elas são executadas. No entanto, o Hibernate garante que a `Query.list(..)` nunca retornará dados antigos, nem retornará dados errados.

It is possible to change the default behavior so that flush occurs less frequently. The `FlushMode` class defines three different modes: only flush at commit time when the Hibernate `Transaction` API is used, flush automatically using the explained routine, or never flush unless `flush()` is called explicitly. The last mode is useful for long running units of work, where a `Session` is kept open and disconnected for a long time (see *Seção 13.3.2, "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();
```

During flush, an exception might occur (e.g. if a DML operation violates a constraint). Since handling exceptions involves some understanding of Hibernate's transactional behavior, we discuss it in *Capítulo 13, Transações e Concorrência* .

11.11. Persistência Transitiva

É um tanto incômodo salvar, deletar ou reanexar objetos individuais, especialmente ao lidar com um grafo de objetos associados. Um caso comum é um relacionamento pai/filho. Considere o seguinte exemplo:

Se os filhos em um relacionamento pai/filho fossem do tipo valor (ex.: coleção de endereços ou strings), seus ciclos de vida dependeriam do pai e nenhuma ação seria requerida para "cascateamento" de mudança de estado. Quando o pai é salvo, os objetos filho de valor são

salvos também, quando o pai é deletado, os filhos também serão deletados, etc. Isto funciona até para operações como remoção de filho da coleção. O Hibernate irá detectar isto e como objetos de valor não podem ter referências compartilhadas, irá deletar o filho do banco de dados.

Agora considere o mesmo cenário com objeto pai e filho sendo entidades, e não de valores (ex.: categorias e ítens, ou cats pai e filho). As entidades possuem seus próprios ciclos de vida, suportam referências compartilhadas (portanto, remover uma entidade da coleção não significa que possa ter sido deletada), e não existe efeito cascata de estado, por padrão, a partir de uma entidade para outras entidades associadas. O Hibernate não implementa *persistência por alcance* por padrão.

Para cada operação básica da sessão do Hibernate, incluindo`persistir()`, `mesclar()`, `salvarOuAtualizar()`, `deletar()`, `bloquear()`, `atualizar()`, `despejar()`, `replicar()`, existe um estilo cascata correspondente. Respectivamente, os estilos cascatas são nomeados `criar`, `mesclar`, `salvar-atualizar`, `deletar`, `bloquiar`, `atualizar`, `despejar`, `replicar`. Se desejar uma operação em cascata junto a associação, você deverá indicar isto no documento de mapeamento. Por exemplo:

```
<one-to-one name="person" cascade="persist"/>
```

Estilo cascata pode ser combinado:

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

Você pode até utilizar `cascade="all"` para especificar que *todas* as operações devem estar em cascata junto à associação. O padrão `cascade="none"` especifica que nenhuma operação deve estar em cascata.

In case you are using annotatons you probably have noticed the `cascade` attribute taking an array of `CascadeType` as a value. The cascade concept in JPA is very is similar to the transitive persistence and cascading of operations as described above, but with slightly different semantics and cascading types:

- `CascadeType.PERSIST`: cascades the persist (create) operation to associated entities persist() is called or if the entity is managed
- `CascadeType.MERGE`: cascades the merge operation to associated entities if merge() is called or if the entity is managed
- `CascadeType.REMOVE`: cascades the remove operation to associated entities if delete() is called
- `CascadeType.REFRESH`: cascades the refresh operation to associated entities if refresh() is called
- `CascadeType.DETACH`: cascades the detach operation to associated entities if detach() is called

- `CascadeType.ALL`: all of the above

Nota

CascadeType.ALL also covers Hibernate specific operations like save-update, lock etc...

A special cascade style, `delete-orphan`, applies only to one-to-many associations, and indicates that the `delete()` operation should be applied to any child object that is removed from the association. Using annotations there is no `CascadeType.DELETE-ORPHAN` equivalent. Instead you can use the attribute `orphanRemoval as seen in` *Exemplo 11.4, "@OneToMany with orphanRemoval"*. If an entity is removed from a `@OneToMany` collection or an associated entity is dereferenced from a `@OneToOne` association, this associated entity can be marked for deletion if `orphanRemoval` is set to true.

Exemplo 11.4. `@OneToMany` with `orphanRemoval`

```
@Entity
public class Customer {
   private Set<Order> orders;

   @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)
   public Set<Order> getOrders() { return orders; }

   public void setOrders(Set<Order> orders) { this.orders = orders; }

   [...]
}

@Entity
public class Order { ... }

Customer customer = em.find(Customer.class, 1l);
Order order = em.find(Order.class, 1l);
customer.getOrders().remove(order); //order will be deleted by cascade
```

Recomendações:

- It does not usually make sense to enable cascade on a many-to-one or many-to-many association. In fact the `@ManyToOne` and `@ManyToMany` don't even offer a `orphanRemoval` attribute. Cascading is often useful for one-to-one and one-to-many associations.
- If the child object's lifespan is bounded by the lifespan of the parent object, make it a *life cycle object* by specifying `cascade="all,delete-orphan"(@OneToMany(cascade=CascadeType.ALL, orphanRemoval=true))`.
- Caso contrário, você pode não precisar realizar a cascata. Mas se você achar que irá trabalhar com o pai e filho juntos com freqüência, na mesma transação, e quiser salvar você mesmo, considere o uso do `cascata="persistir,mesclar,salvar-atualizar"`.

Ao mapear uma associação (tanto uma associação de valor único como uma coleção) com `casca de="all"`, a associação é demarcada como um relacionamento de estilo *parent/child* onde salvar/atualizar/deletar do pai, resulta em salvar/atualizar/deletar do(s) filho(s).

Furthermore, a mere reference to a child from a persistent parent will result in save/update of the child. This metaphor is incomplete, however. A child which becomes unreferenced by its parent is *not* automatically deleted, except in the case of a one-to-many association mapped with `cascade="delete-orphan"`. The precise semantics of cascading operations for a parent/child relationship are as follows:

- Se um pai é passado para `persist()`, todos os filhos são passados para `persist()`
- Se um pai é passado para `merge()`, todos os filhos são passados para `merge()`
- Se um pai for passado para `save()`, `update()` ou `saveOrUpdate()`, todos os filhos passarão para `saveOrUpdate()`
- Se um filho transiente ou desanexado se tornar referenciado pelo pai persistente, ele será passado para `saveOrUpdate()`
- Se um pai for deletado, todos os filhos serão passados para `delete()`
- Se um filho for diferenciado pelo pai persistente, *nada de especial acontece* - a aplicação deve explicitamente deletar o filho se necessário, a não ser que `casca de="delete-orphan"`, nos quais casos o filho "órfão" é deletado.

Finalmente, note que o cascateamento das operações podem ser aplicados a um grafo de objeto em *tempo de chamada* ou em *tempo de limpeza*. Todas as operações, se habilitadas, são colocadas em cascata para entidades associadas atingíveis quando a operação for executada. No entanto, `save-upate` e `delete-orphan` são transitivas para todas as entidades associadas atingíveis durante a limpeza da `Sessão`.

11.12. Usando metadados

O Hibernate requer um modelo muito rico, em 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 os 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] );
    }
}
```

Read-only entities

Importante

Hibernate's treatment of *read-only* entities may differ from what you may have encountered elsewhere. Incorrect usage may cause unexpected results.

When an entity is read-only:

- Hibernate does not dirty-check the entity's simple properties or single-ended associations;
- Hibernate will not update simple properties or updatable single-ended associations;
- Hibernate will not update the version of the read-only entity if only simple properties or single-ended updatable associations are changed;

In some ways, Hibernate treats read-only entities the same as entities that are not read-only:

- Hibernate cascades operations to associations as defined in the entity mapping.
- Hibernate updates the version if the entity has a collection with changes that dirties the entity;
- A read-only entity can be deleted.

Even if an entity is not read-only, its collection association can be affected if it contains a read-only entity.

For details about the affect of read-only entities on different property and association types, see *Seção 12.2, “Read-only affect on property type”*.

For details about how to make entities read-only, see *Seção 12.1, “Making persistent entities read-only”*

Hibernate does some optimizing for read-only entities:

- It saves execution time by not dirty-checking simple properties or single-ended associations.
- It saves memory by deleting database snapshots.

12.1. Making persistent entities read-only

Only persistent entities can be made read-only. Transient and detached entities must be put in persistent state before they can be made read-only.

Hibernate provides the following ways to make persistent entities read-only:

- you can map an entity class as *immutable*; when an entity of an immutable class is made persistent, Hibernate automatically makes it read-only. see *Seção 12.1.1, "Entities of immutable classes"* for details
- you can change a default so that entities loaded into the session by Hibernate are automatically made read-only; see *Seção 12.1.2, "Loading persistent entities as read-only"* for details
- you can make an HQL query or criteria read-only so that entities loaded when the query or criteria executes, scrolls, or iterates, are automatically made read-only; see *Seção 12.1.3, "Loading read-only entities from an HQL query/criteria"* for details
- you can make a persistent entity that is already in the in the session read-only; see *Seção 12.1.4, "Making a persistent entity read-only"* for details

12.1.1. Entities of immutable classes

When an entity instance of an immutable class is made persistent, Hibernate automatically makes it read-only.

An entity of an immutable class can created and deleted the same as an entity of a mutable class.

Hibernate treats a persistent entity of an immutable class the same way as a read-only persistent entity of a mutable class. The only exception is that Hibernate will not allow an entity of an immutable class to be changed so it is not read-only.

12.1.2. Loading persistent entities as read-only

Nota

Entities of immutable classes are automatically loaded as read-only.

To change the default behavior so Hibernate loads entity instances of mutable classes into the session and automatically makes them read-only, call:

```
Session.setDefaultReadOnly( true );
```

To change the default back so entities loaded by Hibernate are not made read-only, call:

```
Session.setDefaultReadOnly( false );
```

You can determine the current setting by calling:

```
Session.isDefaultReadOnly();
```

If Session.isDefaultReadOnly() returns true, entities loaded by the following are automatically made read-only:

- Session.load()
- Session.get()
- Session.merge()
- executing, scrolling, or iterating HQL queries and criteria; to override this setting for a particular HQL query or criteria see *Seção 12.1.3, "Loading read-only entities from an HQL query/criteria"*

Changing this default has no effect on:

- persistent entities already in the session when the default was changed
- persistent entities that are refreshed via Session.refresh(); a refreshed persistent entity will only be read-only if it was read-only before refreshing
- persistent entities added by the application via Session.persist(), Session.save(), and Session.update() Session.saveOrUpdate()

12.1.3. Loading read-only entities from an HQL query/criteria

Nota

Entities of immutable classes are automatically loaded as read-only.

If Session.isDefaultReadOnly() returns false (the default) when an HQL query or criteria executes, then entities and proxies of mutable classes loaded by the query will not be read-only.

You can override this behavior so that entities and proxies loaded by an HQL query or criteria are automatically made read-only.

For an HQL query, call:

```
Query.setReadOnly( true );
```

`Query.setReadOnly( true )` must be called before `Query.list()`, `Query.uniqueResult()`, `Query.scroll()`, or `Query.iterate()`

For an HQL criteria, call:

```
Criteria.setReadOnly( true );
```

`Criteria.setReadOnly( true )` must be called before `Criteria.list()`, `Criteria.uniqueResult()`, or `Criteria.scroll()`

Entities and proxies that exist in the session before being returned by an HQL query or criteria are not affected.

Uninitialized persistent collections returned by the query are not affected. Later, when the collection is initialized, entities loaded into the session will be read-only if Session.isDefaultReadOnly() returns true.

Using `Query.setReadOnly( true )` or `Criteria.setReadOnly( true )` works well when a single HQL query or criteria loads all the entities and intializes all the proxies and collections that the application needs to be read-only.

When it is not possible to load and initialize all necessary entities in a single query or criteria, you can temporarily change the session default to load entities as read-only before the query is executed. Then you can explicitly initialize proxies and collections before restoring the session default.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

setDefaultReadOnly( true );
Contract contract =
   ( Contract ) session.createQuery(
           "from Contract where customerName = 'Sherman'" )
           .uniqueResult();
Hibernate.initialize( contract.getPlan() );
Hibernate.initialize( contract.getVariations() );
Hibernate.initialize( contract.getNotes() );
setDefaultReadOnly( false );
...
tx.commit();
session.close();
```

If Session.isDefaultReadOnly() returns true, then you can use Query.setReadOnly(false) and Criteria.setReadOnly(false) to override this session setting and load entities that are not read-only.

12.1.4. Making a persistent entity read-only

Nota

Persistent entities of immutable classes are automatically made read-only.

To make a persistent entity or proxy read-only, call:

```
Session.setReadOnly(entityOrProxy, true)
```

To change a read-only entity or proxy of a mutable class so it is no longer read-only, call:

```
Session.setReadOnly(entityOrProxy, false)
```

Importante

When a read-only entity or proxy is changed so it is no longer read-only, Hibernate assumes that the current state of the read-only entity is consistent with its database representation. If this is not true, then any non-flushed changes made before or while the entity was read-only, will be ignored.

To throw away non-flushed changes and make the persistent entity consistent with its database representation, call:

```
session.refresh( entity );
```

To flush changes made before or while the entity was read-only and make the database representation consistent with the current state of the persistent entity:

```
// evict the read-only entity so it is detached
session.evict( entity );

// make the detached entity (with the non-flushed changes) persistent
session.update( entity );

// now entity is no longer read-only and its changes can be flushed
s.flush();
```

12.2. Read-only affect on property type

The following table summarizes how different property types are affected by making an entity read-only.

Tabela 12.1. Affect of read-only entity on property types

Property/Association Type	Changes flushed to DB?
Simple	no*

Property/Association Type	Changes flushed to DB?
(*Seção 12.2.1, "Simple properties"*)	
Unidirectional one-to-one Unidirectional many-to-one (*Seção 12.2.2.1, "Unidirectional one-to-one and many-to-one"*)	no* no*
Unidirectional one-to-many Unidirectional many-to-many (*Seção 12.2.2.2, "Unidirectional one-to-many and many-to-many"*)	yes yes
Bidirectional one-to-one (*Seção 12.2.3.1, "Bidirectional one-to-one"*)	only if the owning entity is not read-only*
Bidirectional one-to-many/many-to-one inverse collection non-inverse collection (*Seção 12.2.3.2, "Bidirectional one-to-many/ many-to-one"*)	only added/removed entities that are not read-only* yes
Bidirectional many-to-many (*Seção 12.2.3.3, "Bidirectional many-to-many"*)	yes

* Behavior is different when the entity having the property/association is read-only, compared to when it is not read-only.

12.2.1. Simple properties

When a persistent object is read-only, Hibernate does not dirty-check simple properties.

Hibernate will not synchronize simple property state changes to the database. If you have automatic versioning, Hibernate will not increment the version if any simple properties change.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

// get a contract and make it read-only
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );

// contract.getCustomerName() is "Sherman"
contract.setCustomerName( "Yogi" );
```

```
tx.commit();

tx = session.beginTransaction();

contract = ( Contract ) session.get( Contract.class, contractId );
// contract.getCustomerName() is still "Sherman"
...
tx.commit();
session.close();
```

12.2.2. Unidirectional associations

12.2.2.1. Unidirectional one-to-one and many-to-one

Hibernate treats unidirectional one-to-one and many-to-one associations in the same way when the owning entity is read-only.

We use the term *unidirectional single-ended association* when referring to functionality that is common to unidirectional one-to-one and many-to-one associations.

Hibernate does not dirty-check unidirectional single-ended associations when the owning entity is read-only.

If you change a read-only entity's reference to a unidirectional single-ended association to null, or to refer to a different entity, that change will not be flushed to the database.

Nota

If an entity is of an immutable class, then its references to unidirectional single-ended associations must be assigned when that entity is first created. Because the entity is automatically made read-only, these references can not be updated.

If automatic versioning is used, Hibernate will not increment the version due to local changes to unidirectional single-ended associations.

In the following examples, Contract has a unidirectional many-to-one association with Plan. Contract cascades save and update operations to the association.

The following shows that changing a read-only entity's many-to-one association reference to null has no effect on the entity's database representation.

```
// get a contract with an existing plan;
// make the contract read-only and set its plan to null
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
```

```
contract.setPlan( null );
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );

// contract.getPlan() still refers to the original plan;

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

The following shows that, even though an update to a read-only entity's many-to-one association has no affect on the entity's database representation, flush still cascades the save-update operation to the locally changed association.

```
// get a contract with an existing plan;
// make the contract read-only and change to a new plan
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
Plan newPlan = new Plan( "new plan"
contract.setPlan( newPlan);
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );
newPlan = ( Contract ) session.get( Plan.class, newPlan.getId() );

// contract.getPlan() still refers to the original plan;
// newPlan is non-null because it was persisted when
// the previous transaction was committed;

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

12.2.2.2. Unidirectional one-to-many and many-to-many

Hibernate treats unidirectional one-to-many and many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks unidirectional one-to-many and many-to-many associations;

The collection can contain entities that are read-only, as well as entities that are not read-only.

Entities can be added and removed from the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in the collection if they dirty the owning entity.

12.2.3. Bidirectional associations

12.2.3.1. Bidirectional one-to-one

If a read-only entity owns a bidirectional one-to-one association:

- Hibernate does not dirty-check the association.
- updates that change the association reference to null or to refer to a different entity will not be flushed to the database.
- If automatic versioning is used, Hibernate will not increment the version due to local changes to the association.

Nota

If an entity is of an immutable class, and it owns a bidirectional one-to-one association, then its reference must be assigned when that entity is first created. Because the entity is automatically made read-only, these references cannot be updated.

When the owner is not read-only, Hibernate treats an association with a read-only entity the same as when the association is with an entity that is not read-only.

12.2.3.2. Bidirectional one-to-many/many-to-one

A read-only entity has no impact on a bidirectional one-to-many/many-to-one association if:

- the read-only entity is on the one-to-many side using an inverse collection;
- the read-only entity is on the one-to-many side using a non-inverse collection;
- the one-to-many side uses a non-inverse collection that contains the read-only entity

When the one-to-many side uses an inverse collection:

- a read-only entity can only be added to the collection when it is created;
- a read-only entity can only be removed from the collection by an orphan delete or by explicitly deleting the entity.

12.2.3.3. Bidirectional many-to-many

Hibernate treats bidirectional many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks bidirectional many-to-many associations.

The collection on either side of the association can contain entities that are read-only, as well as entities that are not read-only.

Entities are added and removed from both sides of the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in both sides of the collection if they dirty the entity owning the respective collections.

Transações e Concorrência

O fator 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. 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 de acordo com o 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 procura repetidamente por identificadores e consultas de entidade não consultas 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 sintáxe `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.

13.1. Sessão e escopos de transações

Um `SessionFactory` é objeto threadsafe com um custo alto de criação, compartilhado por todas as threads da aplicação. É criado uma única vez, no início 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. Isto não consome nenhum recurso até ser usado.

Uma transação precisa ser o mais curta possível, para reduzir a disputa pelo bloqueio na base de dados. Transações longas impedirão que sua aplicação escale a carga altamente concorrente. Por isso, não é bom 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 única `Session` do Hibernate gerenciar diversas transações ou este é um o relacionamento um-para-um dos escopos? Quando você deve abrir e fechar uma `Session` e como você demarca os limites da transação? Estas questões estão endereçadas nas seguintes seções.

13.1.1. Unidade de trabalho

First, let's define a unit of work. A unit of work is a design pattern described by Martin Fowler as " [maintaining] a list of objects affected by a business transaction and coordinates the writing out

of changes and the resolution of concurrency problems. ”[PoEAA] In other words, its a series of operations we wish to carry out against the database together. Basically, it is a transaction, though fulfilling a unit of work will often span multiple physical database transactions (see *Seção 13.1.2, “Longas conversações”*). So really we are talking about a more abstract notion of a transaction. The term "business transaction" is also sometimes used in lieu of unit of work.

Primeiro, não use o antipattern *sessão-por-operação*: isto é, não abra e feche uma `Session` para cada simples chamada ao banco de dados em uma única thread. Naturalmente, o mesmo se aplica às transações do banco de dados. As chamadas ao 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 realizar um auto-commit depois de cada instrução 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, aplique a modalidade auto-commit imediatamente. As transações de banco de dados 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, uma vez que muitas transações pequenas são improváveis de executar melhor do que uma unidade de trabalho claramente definida. A última opção é também muito mais sustentável e expandida.

O modelo mais comum em uma aplicação de cliente/servidor multi-usuário é *sessão-por-requisição*. Neste modelo, uma requisição do cliente é enviada ao servidor, onde a camada de persistência do Hibernate é executada. 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 descarregada e fechada. Você usaria também uma única transação de base de dados para servir às requisições dos clientes, iniciando e submetendo-o ao abrir e fechar da `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 gerenciamento integrado da "sessão atual" para simplificar este modelo. Tudo que você tem a fazer é iniciar uma transação quando uma requisição precisa ser processada e terminar 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 recipiente de proxy/interceptação. Um recipiente de EJB é uma maneira padronizada de implementar aspectos cross-cutting, tais como a demarcação da transação em beans de sessão EJB, declarativamente com CMT. Se você se decidir usar demarcação programática de transação, dê preferência à API `Transaction` do Hibernate mostrada mais adiante neste capítulo, para facilidade no uso e portabilidade de código.

Your application code can access a "current session" to process the request by calling `sessionFactory.getCurrentSession()`. You will always get a `Session` scoped to the current database transaction. This has to be configured for either resource-local or JTA environments, see *Seção 2.3, “Sessões Contextuais”*.

Ás 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 renderização separada depois da requisição ter sido processada. Estender a transação até que a renderização da visão esteja completa é fácil de fazer se você implementar seu próprio interceptador. Entretanto, não será fácil se você confiar em EJBs com transações gerenciadas por recipiente, porque uma transação será terminada quando um método de EJB retornar, antes que a renderização de toda visão possa começar. Veja o website e o fórum do Hibernate para dicas e exemplos em torno deste modelo de *Sessão Aberta na Visualização*.

13.1.2. Longas conversações

O modelo 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 da web e corporativas não é aceitável que uma transação atrapalhe uma interação do usuário. Considere o seguinte exemplo:

- A primeira tela de um diálogo se abre e os dados vistos pelo usuário são carregados em uma `Session` e transação de banco de dados particulares. O usuário está livre para modificar os objetos.
- O usuário clica em "Salvar" após 5 minutos e espera suas modificações serem persistidas. O usuário também espera que ele seja 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, uma *conversação* de longa duração (ou *transação da aplicação*). Há muitas maneiras de você 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, uma vez que a disputa do bloqueio não permitiria o escalonameneto da aplicação com o número de usuários concorrentes.

Claramente, 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 parece, 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.
- *Objetos Desanexados*: se você se decidir usar o já discutido pattern *sessão-por-solicitação*, todas as instâncias carregadas estarão no estado destacado durante o tempo em que o

usuário estiver pensando. O Hibernate permite que você re-anexe os objetos e persista as modificações, esse pattern é chamado *sessão-por-solicitação-com-objetos-desanexados*. Utiliza-se versionamento automático para isolar as modificações concorrentes.

- *Sessão Estendida (ou Longa)* A `Session` do Hibernate pode ser desligada da conexão adjacente do JDBC depois que a transação foi submetida, e ser reconectada quando uma nova requisição do cliente ocorrer. Este pattern é conhecido como *sessão-por-conversação* e faz o reatamento uniforme desnecessário. Versionamento automático é usado para isolar modificações concorrentes e a *sessão-por-conversação* geralmente pode ser nivelada automaticamente, e sim explicitamente.

Tanto a *sessão-por-solicitação-com-objetos-desanexados* quanto a *sessão-por-conversação* possuem vantagens e desvantagens. Estas desvantagens serão discutidas mais tarde neste capítulo no contexto do controle de concorrência otimista.

13.1.3. 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`. Portanto, 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 uma `Session` *específica* (ex.: isto está no escopo de uma `Session`), as duas noções são equivalentes e a identidade da JVM para a identidade da base de dados é garantida pelo Hibernate. Entretanto, embora a aplicação possa 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 uma abordagem otimista.

Este caminho deixa o Hibernate e o banco de dados se preocuparem com a concorrência. Ele também fornece uma escalabilidade melhor, garantindo que a identidade em unidades de trabalho single-threaded 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.

No entanto, 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`. Ambas podem ter a mesma identidade na base de dados (ex.: elas representam a mesma linha), mas a identidade da JVM não é, por definição, garantida para instâncias em estado desacoplado. O desenvolvedor tem que substituir 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`, a mudança do hashcode quebrará o contrato do `Set`. As funções 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 característica do Hibernate, mas simplesmente a maneira como a identidade e a igualdade do objeto de Java têm que ser implementadas.

13.1.4. Edições comuns

Nunca use o anti-patterns *sessão-por-usuário-sessão* ou *sessão-por-aplicação* (naturalmente, existem exceções raras para essa regra). Note que algumas das seguintes edições podem também aparecer com modelos 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, beans de sessão, 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 deverá considerar sincronizar o acesso a sua sessão do HTTP. Caso contrário, um usuário que clica em recarga rápido demais, pode usar o mesmo `Session` em duas threads executando simultaneamente.

- 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 limitada pela aplicação, você tem que parar a aplicação. Fazer o rollback na transação no banco de dados não retorna seus objetos do negócio ao estado 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.

- The `Session` caches every object that is in a persistent state (watched and checked for dirty state by Hibernate). If you keep it open for a long time or simply load too much data, it will grow endlessly until you get an OutOfMemoryException. One solution is to call `clear()` and `evict()` to manage the `Session` cache, but you should consider a Stored Procedure if you need mass data operations. Some solutions are shown in *Capítulo 15, Batch processing*. Keeping a `Session` open for the duration of a user session also means a higher probability of stale data.

13.2. 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 acostumados ao 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 sempre demarcar 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, precisa ajustar manualmente os limites das transaçãos, ou seja, começar, submeter ou efetar rollback nas transações ele mesmo. Um ambiente gerenciado fornece transações gerenciadas por recipiente (CMT), com um conjunto da transações definido declarativamente em descritores de implementação de beans de sessão EJB, por exemplo. A demarcação programática é portanto, não mais necessária.

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 use BMT ao invés 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 implementação. Esta API é realmente opcional, mas nós encorajamos fortemente seu uso a menos que você esteja em um bean de sessão CMT.

Geralmente, finalizar uma `Session` envolve quatro fases distintas:

- liberar a sessão
- submeter a transação
- fechar a sessão
- tratar as exceções

A liberação 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.

13.2.1. 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 (ex.: não DataSource) dos quais o Hibernate obtém as conexões assim que necessitar. 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()` da `Session()` explicitamente. A chamada ao `commit()` dispara automaticamente a sincronização para a sessão, dependendo do *Seção 11.10, "Limpando a Sessão"*. 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 gerenciados e de JTA.

Uma solução muito mais flexível é o gerenciamento de contexto "sessão atual" da construção interna 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 "topo". 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`.

13.2.2. Usando JTA

Se sua camada de persistência funcionar em um servidor de aplicação (por exemplo, dentro dos beans de sessão EJB), cada conexão da fonte de dados 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 transações de bean gerenciado (BMT) o Hibernate dirá ao servidor de aplicação para começar e para terminar uma transação de BMT se você usar a `Transaction` API. 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 uma `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 implementação de beans de sessão, não programaticamente, conseqüentemente, o código é reduzido a:

```
// CMT idiom
 Session sess = factory.getCurrentSession();

 // do some work
 ...
```

Em um CMT/EJB, até mesmo um rollback acontece automaticamente, desde que uma exceção `RuntimeException` não tratável seja lançada por um método de um bean de sessão que informa ao recipiente ajustar a transação global ao rollback. *Isto significa que você não precisa mesmo usar a API `Transaction` do Hibernate com BMT ou CMT e você obterá a propagação automática da Sessão "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 bean de sessão CMT, 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 está 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 das aplicações podem facilmente evitar o uso do `scroll()` ou do `iterate()` em todo código provindo do JTA ou do CMT.

13.2.3. Tratamento de Exceção

Se a `Session` levantar uma exceção, incluindo qualquer `SQLException`, você deverá 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 não verificada. Ela não constava em versões mais antigas de 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 não verificadas 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 alguma outra ação apropriada é feita). Note que Hibernate pode também lançar outras exceções não verificadas que não sejam um `HibernateException`. Estas, também são, irrecuperáveis e uma ação apropriada deve ser tomada.

O Hibernate envolve `SQLException`s lançadas ao interagir com a base de dados em um `JDBCException`. Na realidade, o Hibernate tentará converter a exceção em uma subclasse mais significativa da `JDBCException`. A `SQLException` subjacente está sempre disponível através de `JDBCException.getCause()`. O Hibernate converte a `SQLException` em uma subclasse `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 sintáxe 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 está inclusa em nenhuma das outras categorias.

13.2.4. Tempo de espera de Transação

O tempo de espera de transação é uma característica extremamente importante fornecida por um ambiente gerenciado como EJB e que nunca é fornecido pelo código não gerenciado. Os tempos de espera de transação asseguram que nenhuma transação retenha indefinidamente recursos enquanto não retornar 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 consultas do banco de dados com imensos resultados definidos sejam limitados pelo tempo de espera. Em um ambiente gerenciado, o Hibernate pode delegar o tempo de espera 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 bean CMT, onde o tempo de espera das transações deve ser definido declaradamente.

13.3. Controle de concorrência otimista

O único caminho que é consistente com a elevada concorrência e escalabilidade é o controle de concorrência otimista com versionamento. A checagem de versão usa número de versão, ou carimbo de hora (timestamp), 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 conversações longas, mas a checagem de versão também tem o benefício de impedir atualizações perdidas em únicas transações.

13.3.1. 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 por 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 o caminho mais similar à entidade EJBs.

```
// 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-lá automaticamente durante a liberação se a entidade estiver alterada.

Claro, se você estiver operando em um ambiente de baixa concorrência de dados e não precisar da checagem de versão, você pode usar este caminho e apenas pular a checagem de versão. Nesse caso, o *último commit realizado* é a estratégia padrão para suas conversações longas. Tenha em mente que isto pode confundir os usuários da aplicação, como também poderão ter atualizações perdidas sem mensagens de erro ou uma possibilidade de ajustar mudanças conflitantes.

Claro que, a checagem manual da versão é somente possível em circunstâncias triviais e não para a maioria de aplicações. Freqüentemente, os gráficoscompletos 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.

13.3.2. Sessão estendida e versionamento automático

Uma única instância de `Session` e suas instâncias persistentes são usadas para a conversação inteira, isto é conhecido como *sessão-por-conversação*. O Hibernate verifica versões da instância no momento da liberação, 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 adjacente enquanto espera a interação do usuário. Este caminho é o mais eficiente em termos de acesso a bancos de dados. A aplicação não precisa se preocupar com a checagem de versão ou com as instâncias destacadas reatadas, nem precisa recarregar instâncias a 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 a `Session` já foi carregada. Ao começar uma nova transação ou uma sessão velha, você obterá uma conexão nova e reiniciará a sessão. Submeter uma transação implica em desconectar uma sessão da conexão JDBC e retornar à conexão ao pool. Após a reconexão, para forçar uma checagem de versão em dados que você não esteja atualizando, você poderá chamar `Session.lock()` com o `LockMode.READ` em todos os objetos que possam ter sido atualizados por uma outra transação. Você não precisa bloquear nenhum dado que você *está* atualizando. 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. Por isso, somente esta última transação incluiria a operação `flush()` e então também iria `close()` a sessão para terminar a conversação.

Este modelo é problemático se a `Session` for demasiadamente grande para ser armazenada durante o tempo de espera do usuário (por exemplo uma `HttpSession` deve ser mantida o menor possível). Como a `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.

Nota

Note que versões mais atuais de Hibernate requerem a desconexão e reconexão explícitas de uma `Session`. Estes métodos são desatualizados, pois o início e término de uma transação têm o mesmo efeito.

Note também que você deve manter a `Session` desconectada, fechada para a camada de persistência. Ou seja, use um bean de sessão com estado EJB para prender a `Session` em um ambiente de três camadas. Não transfira à camada web, ou até serializá-lo para uma camada separada, para armazená-lo no `HttpSession`.

O modelo da sessão estendida, ou *sessão-por-conversação*, 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.

13.3.3. 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 uma outra `Session` e as reata 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 a liberação, 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.

13.3.4. Versionamento automático customizado

Você pode desabilitar o incremento da versão automática de Hibernate para propriedades e coleções particulares, configurando a função de mapeamento `optimistic-lock` para `false`. O Hibernate então, não incrementará mais versões se a propriedade estiver modificada.

Os esquemas da base de dados legado são freqüentemente estáticos e não podem ser modificados. Ou então, outras aplicações puderam também acessar a mesma base de dados

e não sabem tratar a versão dos números ou carimbos de hora. 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 carimbo de hora 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 o Hibernate puder comparar o estado velho e novo (ex.: se você usar uma única `Session` longa e não uma sessão-por-solicitação-com-objetos-desanexados).

Às vezes a modificação concorrente pode ser permitida, desde que as mudanças realizadas não se sobreponham. Se você configurar `optimistic-lock="dirty"` ao mapear o `<class>`, o Hibernate comparará somente campos modificados durante a liberação.

Em ambos os casos, com as colunas de versão/carimbo de hora dedicados com comparação de campo cheio/sujo, o Hibernate usa uma única instrução `UPDATE`, com uma cláusula `WHERE` apropriada, por entidade para executar a checagem da versão e atualizar a informação. Se você usar a persistência transitiva para cascatear o reatamento das entidades associadas, o Hibernate pode executar atualizações desnecessárias. Isso não é geralmente um problema, mas os triggers *em atualizações* num banco de dados pode ser executado mesmo quando nenhuma mudança foi feita nas instâncias desanexadas. Você pode customizar este comportamento configurando `selecionar-antes-de atualizar="verdadeiro"` no mapeamento `<class>`, forçando o Hibernate a fazer um `SELECT` nas instâncias para assegurar-se de que as mudanças realmente aconteceram, antes de atualizar a linha.

13.4. Bloqueio Pessimista

Não ha intenção alguma que usuários gastem muitas horas se preocupando com suas estratégias de bloqueio. Geralmente, é o bastante especificar um nível de isolamento para as conexões JDBC e então deixar simplesmente o banco de dados fazer todo o trabalho. Entretanto, os usuários avançados podem às vezes desejar obter bloqueios pessimistas exclusivos, ou re-obter bloqueios no início de uma nova transação.

O Hibernate usará sempre o mecanismo de bloqueio da base de dados, nunca bloquiar objetos na memória.

A classe `LockMode` define os diferentes níveis de bloqueio que o Hibernate pode adquirir. Um bloqueio 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 essa sintáxe.
- `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 de Leitura Repetida ou isolamento Serializável. Pode ser readquirido explicitamente pelo usuário.

- `LockMode.NONE` representa a ausência do bloqueio. Todos os objetos mudam para esse estado de bloqueio no final da `Transaction`. Objetos associados com a sessão através do método `update()` ou `saveOrUpdate()` também são inicializados com esse modo de bloqueio.

O bloqueio obtido "explicitamente pelo usuário" se dá nas seguintes formas:

- Uma chamada a `Session.load()`, especificando o `LockMode`.
- Uma chamada à `Session.lock()`.
- Uma chamada à `Query.setLockMode()`.

Se uma `Session.load()` é invocada com `UPGRADE` ou `UPGRADE_NOWAIT`, e o objeto requisitado ainda não foi carregado pela sessão, o objeto é carregado usando `SELECT ... FOR UPDATE`. Se `load()` for chamado para um objeto que já foi carregado com um bloqueio menos restritivo que o novo bloqueio solicitado, o Hibernate invoca o método `lock()` para aquele objeto.

O `Session.lock()` executa uma verificação no número da versão se o modo de bloqueio 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 modo de bloqueio solicitado, o Hibernate usará um modo alternativo apropriado, ao invés de lançar uma exceção. Isso garante que a aplicação seja portátil.

13.5. Modos para liberar a conexão

O comportamento legado do Hibernate 2.x referente ao gerenciamento da conexão via JDBC era que a `Session` precisaria obter uma conexão quando ela precisasse pela primeira vez e depois manteria a conexão enquanto a sessão não fosse fechada. O Hibernate 3.x introduz a idéia de modos para 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. As 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`: é o modo legado descrito acima. A sessão do Hibernate obtém a conexão quando precisar executar alguma operação JDBC pela primeira vez e mantém 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 como liberação agressiva): informa que a conexão deve ser liberada após a execução de cada instrução. A liberação agressiva não ocorre se a instrução deixa pra trás algum recurso aberto associado com a sessão obtida. Atualmente, a única situação em que isto ocorre 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. Segue abaixo os valores possíveis:

- `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, pois 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 não 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. Além disso, 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 tanto readquirir a mesma conexão JDBC adjacente todas as vezes que chamarmos `ConnectionProvider.getConnection()` quanto em um ambiente auto-commit, onde não importa se voltamos para a mesma conexão.

Interceptadores e Eventos

É muito útil quando a aplicação precisa reagir a certos eventos que ocorrem dentro do Hibernate. Isso permite a implementação de certas funções genéricas, assim como permite estender as funcionalidades do Hibernate.

14.1. Interceptadores

A interface `Interceptor` permite fornecer informações da sessão para o aplicativo, permitindo que o aplicativo inspecione e/ou manipule as propriedades de um objeto persistente antes de ser salvo, atualizado, excluído ou salvo. Pode ser usado para gerar informações de auditoria. Por exemplo, o seguinte `Interceptor` ajusta a função automaticamente `createTimestamp` quando um `Auditable` é criado e atualiza a função `lastUpdateTimestamp` quando um `Auditable` é atualizado.

Você pode implementar `Interceptor` diretamente ou pode estender `EmptyInterceptor`.

```
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 se apresentam de duas formas: `Session`-scoped e `SessionFactory`-scoped.

Um interceptador delimitado 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 da `SessionFactory`-scoped 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. Cetifique-se de não armazenar funções de estado específicos da sessão, pois, provavelmente, múltiplas sessões irão utilizar esse interceptador simultaneamente.

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

14.2. Sistema de Eventos

Se você precisar 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 arquivo deconfiguração 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 devem ser considerados singletons. Isto significa que 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 equivalentes (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 especificado. A configuração declarativa através do arquivo de propriedades 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 instância. Se o mesmo nome da classe for utilizado em vários elementos `<listener/>`, cada um resultará em uma instância separada dessa classe. Se você tem a necessidade de compartilhar uma instância de um listener entre diversos tipos de listeners você deve registrar o listener programaticamente.

Mas, por quê 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.

14.3. Segurança declarativa do Hibernate

Geralmente a segurança declarativa nos aplicativos do Hibernate é gerenciada em uma camada de fachada de sessão. Agora o Hibernate3 permite certas ações serem aceitas através do JACC e autorizadas através do JAAS. Esta é uma funcionalidade opcional construída em cima da arquitetura do evento.

Primeiro, você precisa configurar um evento listener apropriado, para possibilitar o uso da autorização 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"/>
```

Note que `<listener type="..." class="..."/>` é somente um atalho para `<event type="..."><listener class="..."/></event>` quando existir somente um listener para um tipo de evento em particular.

Depois disso, ainda em `hibernate.cfg.xml`, vincule as permissões aos papéis:

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions="*"/>
```

Os nomes das funções são as funções conhecidas pelo seu provedor JACC.

Batch processing

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 com um `OutOfMemoryException` em algum lugar próximo a linha 50.000. Isso ocorre devido ao fato do Hibernate fazer cache de todas as instâncias de `Customer` inseridas num cachê em nível de sessão. Nós demonstraremos neste capitulo como evitar este problema.

Entretanto, se você vai realizar processamento em 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, por exemplo):

```
hibernate.jdbc.batch_size 20
```

Note que o Hibernate desabilita o loteamento de inserção no nível JDBC de forma transparente se você utilizar um gerador de identificador `identiy`.

Você também pode querer rodar esse tipo de processamento em lotes com o cache secundário completamente desabilitado:

```
hibernate.cache.use_second_level_cache false
```

Mas isto não é absolutamente necessário, desde que possamos ajustar o `CacheMode` para desabilitar a interação com o cache secundário.

15.1. Inserção em 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();
```

15.2. Atualização em lotes

Para recuperar e atualizar informações a mesma idéia é válida. Além disso, pode precisará usar o `scroll()` para usar recursos no lado do servidor em consultas que retornem 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();
```

15.3. A interface de Sessão sem Estado

Como forma alternativa, o Hibernate provê uma API orientada à comandos, que pode ser usada para transmitir um fluxo de dados de e para o banco de dados na forma de objetos desanexados. Um `StatelessSession` não tem um contexto persistente associado e não fornece muito das semânticas de alto nível para controle do ciclo de vida. Especialmente uma Sessão sem Estado não implementa um cachê primário e nem interage com o cache secundário ou cachê de consulta. Ela não implementa uma gravação temporária transacional ou checagem suja automática. Operações realizadas usando uma sessão sem estado não fazem nenhum tipo de cascata com as instâncias associadas. As coleções são ignoradas por uma Sessão sem Estado. Operações realizadas com uma Sessão sem Estado ignoram a arquitetura de eventos e os interceptadores. As sessões sem estado são vulneráveis aos efeitos do alias dos dados,

devido à falta do cachê primário. Uma Sessão sem Estado é uma abstração de baixo nível, muito mais próxima do JDBC adjacente.

```
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 instâncias de `Customer` retornadas pela consulta, são imediatamente desvinculadas. Elas nunca serão associadas à um contexto persistente.

As operações `insert(), update()` e `delete()` definidas pela interface `StatelessSession` são considerados operações diretas no banco de dados. Isto resulta em uma execução imediata de comandos SQL `INSERT, UPDATE` ou `DELETE` respectivamente. Dessa forma, eles possuem uma semântica bem diferente das operações `save(), saveOrUpdate()` ou `delete()` definidas na interface `Session`.

15.4. Operações no estilo DML

Como já discutido anteriormente, o mapeamento objeto/relacional automático e transparente é adquirido com a gerência do estado do objeto. Com isto o estado daquele objeto fica disponível na memória. Isto significa que a manipulação de dados (usando as instruções SQL `Data Manipulation Language` (SQL-style DML): `INSERT`, `UPDATE`, `DELETE`) diretamente no banco de dados não irá afetar o estado registrado em memória. Entretanto, o Hibernate provê métodos para executar instruções de volume de SQL-style DML, que são totalmente executados com HQL (Hibernate Query Language - Linguagem de Consulta Hibernate) (*HQL*).

A pseudo-sintaxe para instruções `UPDATE` e `DELETE` é: Algumas observações: `( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?`.

Alguns pontos a serem destacados:

- Na cláusula from, a palavra chave FROM é opcional;
- Somente uma entidade pode ser chamada na cláusula from. Isto pode, opcionalmente, ser um alias. Se o nome da entidade for um alias, então qualquer referência de propriedade deve ser qualificada usando esse alias. Caso o nome da entidade não for um alias, então será ilegal qualquer das referências de propriedade serem qualificadas.

- Nenhum *joins*, tanto implícito ou explícito, pode ser especificado em uma consulta de volume HQL. As Sub-consultas podem ser utilizadas na cláusula onde, em que as subconsultas podem conter uniões.
- A clausula onde também é opcional.

Como exemplo para executar um HQL `UPDATE`, use o método `Query.executeUpdate()`. O método ganhou o nome devido à 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();
```

As instruções do HQL `UPDATE` por padrão não afetam o *version* ou os valores de propriedade *timestamp* para as entidades afetadas, de acordo com a especificação EJB3. No entanto, você poderá forçar o Hibernate a redefinir corretamente os valores de propriedade `version` ou `timestamp` usando um `versioned update`. Para tal, adicione uma palavra chave `VERSIONED` após a palavra chave `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();
```

Note que os tipos de versões padronizadas, `org.hibernate.usertype.UserVersionType`, não são permitidos junto às instruções `update versioned`.

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 número 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 de volume HQL pode resultar em várias instruções SQL atuais a serem executadas (por exemplo, no caso de subclasses unidas). O número retornado indica a quantidade real de entidades afetadas pela instrução. Voltando ao exemplo da subclasse unida, 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 de subclasses unidas num nível hierárquico imediatamente abaixo.

A pseudo-sintáxe 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 ao `column specification` do comando SQL `INSERT`. Para entidades envolvidas em mapeamentos, apenas as propriedades definidas diretamente em nível da classe podem ser usadas na properties_list. Propriedades da superclasse não são permitidas e as propriedades da subclasse não fazem sentido. Em outras palavras, os comandos `INSERT` não são polimórficos.
- selecionar_instruções pode ser qualquer consulta de seleção HQL válida, desde que os tipos de retorno sejam compatíveis com os tipos esperados pela inserção. Atualmente, isto é verificado durante a compilação da consulta, ao invés de permitir que a verificação chegue ao banco de dados. Entretanto, perceba que isso pode causar problemas entre os `Tipos` de Hibernate que são *equivalentes* e não *iguais*. Isso pode causar problemas nas combinações entre a propriedade definida como `org.hibernate.type.DateType`e uma 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 instrução insert oferece duas opções. Você pode especificar qualquer propriedade id explicitamente no properties_list (em alguns casos esse valor é obtido diretamente da instrução select) ou pode omitir do properties_list (nesse caso, um valor gerado é usado). Essa última 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" irá causar um exceção durante a etapa de análise. Note que para a finalidade 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 os seus valores.
- Para propriedades mapeadas como `version` ou `timestamp`, a instrução insert lhe oferece duas opções. Você pode especificar a propriedade na properties_list, nesse caso o seu valor é obtido

a partir da instrução select correspondente, ou ele pode ser omitido da properties_list (neste caso utiliza-se o `seed value` definido pela classe `org.hibernate.type.VersionType`).

Segue abaixo o 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();
```

HQL: A Linguagem de Consultas do Hibernate

O Hibernate vem com uma poderosa linguagem de consulta (HQL) que é muito parecida com o SQL. No entanto, comparado com o SQL o HQL é totalmente orientado à objetos, e compreende noções de herança, polimorfismo e associações.

16.1. Diferenciação de maiúscula e minúscula

As Consultas não diferenciam maiúscula de minúscula, exceto pelo nomes das classes e propriedades Java. Portanto, `SeLeCT` é o mesmo que `sELEct` que é 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 acreditam que com letras maiúsculas as consultas ficam mais legíveis, mas nós acreditamos que este formato não é apropriado para o código Java.

16.2. A cláusula from

A consulta mais simples possível do Hibernate é a seguinte:

```
from eg.Cat
```

Isto simplesmente retornará todas as instâncias da classe `eg.Cat`. Geralmente não precisamos qualificar o nome da classe, uma vez que o `auto-import` é o padrão. Por exemplo:

```
from Cat
```

Com o objetivo de referir-se ao `Cat` em outras partes da consulta, você precisará determinar um *alias*. Por exemplo:

```
from Cat as cat
```

Essa consulta atribui um alias a `cat` para as instâncias de `Cat`, portanto poderemos usar esse alias mais tarde na consulta. A palavra chave `as` é opcional. Você também pode escrever assim:

```
from Cat cat
```

Classes múltiplas podem ser envolvidas, resultando em um produto cartesiano ou união "cruzada".

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

É considerada uma boa prática nomear alias de consulta, utilizando uma letra minúscula inicial, consistente com os padrões de nomeação Java para variáveis locais (ex.: `domesticCat`).

16.3. Associações e uniões

Podemos também atribuir aliases em uma entidade associada, ou mesmo em elementos de uma coleção de valores, usando uma `join`. Por exemplo:

```
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 uniões suportados foram inspirados no ANSI SQL:

- `inner join`
- `left outer join`
- `right outer join`
- `união completa` (geralmente não é útil)

As construções `inteiro`, `união esquerda externa` e `união direita externa` podem ser abreviadas.

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

Você pode fornecer condições extras de união usando a palavra chave do HQL `with`.

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight
> 10.0
```

A "fetch" join allows associations or collections of values to be initialized along with their parent objects using a single select. This is particularly useful in the case of a collection. It effectively overrides the outer join and lazy declarations of the mapping file for associations and collections. See *Seção 21.1, “Estratégias de Busca ”* for more information.

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

Geralmente, uma união de busca não precisa atribuir um alias, pois o objeto associado não deve ser usado na cláusula `where` (ou em qualquer outra cláusula). Também, os objetos associados não são retornados diretamente nos resultados da consulta. Ao invés disso, eles devem ser acessados usando o objeto pai. A única razão pela qual precisariamos de um alias é quando fazemos uma união de busca recursivamente em uma coleção adicional:

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

Observe que a construção `busca` não deve ser usada em consultas 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 contém duplicidade devido à busca das coleções, então o número de linhas pode não ser o que você espera. A `Fetch` não deve ser usada junto com uma condição `with`. É possível que seja criado um produto cartesiano pela busca de união em mais do que uma coleção em uma consulta, então tome cuidado nesses casos. Uma busca de união em várias coleções pode trazer resultados inesperados para mapeamentos do tipo bag, tome cuidado na hora de formular consultas como essas. Finalmente, observe o seguinte, a `busca de união completa` e `busca de união direita` não são importantes.

Se estiver usando o nível de propriedade busca lazy (com instrumentação de bytecode), é possível forçar o Hibernate a buscar as propriedades lazy imediatamente na primeira consulta, usando `buscar todas as propriedades .`

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

16.4. Formas de sintáxe de uniões

O HQL suporta duas formas de associação para união: `implícita` e `explícita`.

As consultas apresentadas na seção anterior usam a forma `explícita`, onde a palavra chave união é explicitamente usada na cláusula from. Essa é a forma recomendada.

A forma `implícita` não usa a palavra chave "união". Entretanto, as associações são "diferenciadas" usando pontuação ("." - dot-notation). Uniões`implícitas` podem aparecer em qualquer uma das cláusulas HQL. A união `implícita` resulta em declarações SQL que contém uniões inteiras.

```
from Cat as cat where cat.mate.name like '%s%'
```

16.5. Referência à propriedade do identificador

Geralmente, existem duas formas para se referir à propriedade do indentificador de uma entidade:

- A propriedade especial (em letra minúscula) `id` pode ser usada para se referir à propriedade do identificador de uma entidade *considerando que a entidade não define uma propriedade não identificadora chamada id*.
- Se a entidade definir a propriedade do identificador nomeada, você poderá usar este nome de propriedade.

As referências à composição das propriedades do identificador seguem as mesmas regras de nomeação. Se a entidade tiver uma propriedade de não identificador chamada id, a composição da propriedade do identificador pode somente ser referenciada pelo seu nome definido. Do contrário, uma propriedade especial `id` poderá ser usada para referenciar a propriedade do identificador.

Importante

Observe: esta ação mudou completamente na versão 3.2.2. Nas versões anteriores o `id` *sempre* referia-se à propriedade do identificador não importando seu nome atual. Uma ramificação desta decisão era que as propriedades do não identificador de chamadas `id` nunca poderiam ser referenciadas nas consultas do Hibernate.

16.6. A cláusula select

A cláusula `select` seleciona quais objetos e propriedades retornam no resultado da consulta. Considere:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

A consulta selecionará `mate`s (parceiros), de outros `Cat`s. Atualmente, podemos expressar a consulta de forma mais compacta como:

```
select cat.mate from Cat cat
```

As consultas 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
```

As consultas podem retornar múltiplos objetos e/ou propriedades como uma matriz 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 - considerando que a classe `Family` tenha um construtor apropriado - como um objeto Java typesafe atual:

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

Pode-se designar alias à expressões selecionadas usando `as`:

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

Isto é bem mais útil quando usado junto `comselecione novo mapa`:

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

Esta consulta retorna um `Mapa` de referências para valores selecionados.

16.7. Funções de agregação

As consultas 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 aritméticos, concatenação e funções SQL reconhecidas na cláusula 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 que no SQL.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

16.8. Pesquisas Polimórficas

A consulta:

```
from Cat as cat
```

retorna instâncias não só de `Cat`, mas também de subclasses como `DomesticCat`. As consultas do Hibernate podem nomear qualquer classe Java ou interface na cláusula `from`. A consulta retornará instâncias de todas as classes persistentes que extendam a determinada classe ou implemente a determinada interface. A consulta a seguir, poderia retornar todos os objetos persistentes:

```
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 cláusula `order by` não ordena corretamente todo o resultado. Isso também significa que você não pode chamar essas consultas usando `consulta.scroll()`.

16.9. A cláusula where

A cláusula `where` permite estreitar a lista de instâncias retornadas. Se não houver referência alguma, pode-se referir à 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'
```

Isto retorna instâncias de `Cat` com nome ‘Fritz’.

A seguinte consulta:

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

retornará todas as instâncias de `Foo`, para cada um que tiver uma instância de `bar` com a propriedade `date` igual a propriedade `startDate` de `Foo`. Expressões de caminho compostas fazem da cláusula `where`, extremamente poderosa. Consideremos:

```
from Cat cat where cat.mate.name is not null
```

Esta consulta traduz para uma consulta SQL com uma tabela (inner) união. Por exemplo:

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

resultaria numa consulta que necessitasse de união de quatro tabelas, no SQL.

O operador = pode ser usado para comparar não apenas propriedades, mas também instâncias:

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

The special property (lowercase) `id` can be used to reference the unique identifier of an object. See *Seção 16.5, “Referência à propriedade do identificador ”* for more information.

```
from Cat as cat where cat.id = 123

from Cat as cat where cat.mate.id = 69
```

A segunda consulta é eficiente e não requer nenhuma união de tabelas.

As propriedades de identificadores compostas também podem ser usadas. Considere o seguinte exemplo onde `Person` possui identificadores compostos que consistem 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 consulta não precisa de nenhuma união de tabela.

See *Seção 16.5, "Referência à propriedade do identificador "* for more information regarding referencing identifier properties)

Da mesma forma, a propriedade especial `class` acessa o valor discriminador da instância, no caso de persistência polimórfica. O nome de uma classe Java inclusa em uma cláusula where, será traduzida para seu valor discriminante.

```
from Cat cat where cat.class = DomesticCat
```

You can also use components or composite user types, or properties of said component types. See *Seção 16.17, "Componentes"* for more information.

Um tipo "any" possui as propriedades `id` e `class` especiais, nos permitindo expressar uma união 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 à valores de colunas de banco de dados completamente diferentes, na consulta acima.

16.10. Expressões

As expressões permitidas na cláusula `where` incluem o seguinte:

- operadores matemáticos: `+, -, *, /`
- operadores de comparação binários: `=, >=, <=, <>, !=, like`
- operadores lógicos `and, or, not`
- Parênteses `( )` que indica o agrupamento
- `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()` e `current_timestamp()`
- `second(...)`, `minute(...)`, `hour(...)`, `day(...)`, `month(...)` e `year(...)`
- qualquer função ou operador definidos pela EJB-QL 3.0: `substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()`
- `coalesce()` and `nullif()`
- `str()` para converter valores numéricos ou temporais para uma string de leitura
- `cast(... as ...)`, onde o segundo argumento é o nome do tipo hibernate, e`extract(... from ...)` se ANSI `cast()` e `extract()` é suportado pelo banco de dados adjacente
- A função HQL `index()` , que se aplicam às referências 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 `índices` que podem ser quantificadas usando `some, all, exists, any, in`.
- Qualquer função escalar suportada pelo banco de dados como `sign()`, `trunc()`, `rtrim()` e `sin()`
- Parâmetros posicionais ao estilo JDBC `?`
- Parâmetros nomeados `:name`, `:start_date` e `:x1`
- Literais SQL `'foo'`, `69`, `6.66E+2`, `'1970-01-01 10:00:01.0'`
- Constantes Java `final estático públicoex: 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' )
```

As formas negativas podem ser escritas conforme segue abaixo:

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

Da mesma forma, `is null` e `is not null` podem ser usados para testar valores nulos.

Booleanos podem ser facilmente usados em expressões, declarando as substituições da consulta HQL, na configuração do Hibernate:

```
<property name="hibernate.query.substitutions"
>true 1, false 0</property
>
```

Isso irá substituir as palavras chave `true` e `falsepelos` 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`. Igualmente, pode-se referir aos elementos máximo e mínimo de uma coleção de tipos básicos usando as funções `minelement` e `maxelement`. Por exemplo:

```
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 `índices` de funções) ou o resultado de uma subconsulta (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 - `tamanho`, `elementos`, `índices`, `minindex`, `maxindex`, `minelement`, `maxelement` – só podem ser usados na cláusula where do Hibernate3.

Elementos de coleções com índice (matriz, listas, mapas) podem ser referenciadas pelo índice (apenas na cláusula 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-para-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 podem ser usadas:

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

Se ainda não estiver totalmente convencido, pense o quão maior e menos legível poderia ser a consulta 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: algo como:

```
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
    )
```

16.11. A cláusula ordenar por

A lista retornada pela consulta pode ser ordenada por qualquer propriedade da classe ou componentes retornados:

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

As opções `asc` ou `desc` indicam ordem crescente ou decrescente, respectivamente.

16.12. A cláusula agrupar por

Uma consulta que retorne valores agregados, podem ser agrupados por qualquer propriedade de uma classe ou componentes retornados:

```
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 cláusula `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 cláusulas `having` e `order by`, se suportadas pelo banco de dados subjacentes (ex: não no MeuSQL).

```
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 cláusula `group by` ou `order by` podem conter expressões aritméticas. O Hibernate também não expande atualmente uma entidade agrupada, portanto você não pode escrever `group by cat` caso todas as propriedades do `cat` não estiverem agregadas. Você precisa listar claramente todas as propriedades não-agregadas.

16.13. Subconsultas

Para bancos de dados que suportam subseleções, o Hibernate suporta subconsultas dentro de consultas. Uma subconsulta precisa estar entre parênteses (normalmente uma chamada de função agregada SQL). Mesmo subconsultas co-relacionadas (subconsultas que fazem referência à alias de outras consultas), 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 subconsultas podem aparecer apenas dentro de cláusulas select ou where.

Note that subqueries can also utilize `row value constructor` syntax. See *Seção 16.18, "Sintáxe do construtor de valores de linha"* for more information.

16.14. Exemplos de HQL

As consultas do Hibernate, podem ser muito poderosas e complexas. De fato, o poder da linguagem de consulta é um dos pontos principais na distribuição do Hibernate. Aqui temos algumas consultas de exemplo, muito similares a consultas usadas em um projeto recente. Note que a maioria das consultas que você irá escrever, são mais simples que estas.

A consulta a seguir retorna o id de ordenar, número de ítens e o valor total do ordenar para todos os ordenar não pagos para um cliente particular e valor total mínimo dado, ordenando os resultados por valor total. Para determinar os preços, utiliza-se o catálogo atual. A consulta SQL resultante, usando tabelas `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` e `PRICE`, têm quatro uniões inteiras e uma subseleção (não correlacionada).

```
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! Na verdade, na vida real, eu não sou muito afeiçoado à subconsultas, então minha consulta 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 consulta conta o número de pagamentos em cada status, excluindo todos os pagamentos no status `AWAITING_APPROVAL`, onde a mais recente mudança de status foi feita pelo usuário atual. Traduz-se para uma consulta SQL `com` duas uniões inteiras e uma subseleção correlacionada, nas tabelas `PAYMENT`, `PAYMENT_STATUS` e `PAYMENT_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 coleção `statusChanges` como um List, ao invés de um Set, a consulta 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 consulta usa a função `isNull()` do Servidor MS SQL, para retornar todas as contas e pagamentos não efetuados para a organização, para aquele que o usuário atual pertencer. Traduz-se para uma consulta SQL `com` três uniões inteiras, uma união externa e uma subseleção 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 eliminar a subseleção (correlacionada).

```
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
```

16.15. Atualização e correção em lote

HQL now supports `update`, `delete` and `insert ... select ...` statements. See *Seção 15.4, "Operações no estilo DML"* for more information.

16.16. Dicas & Truques

Pode-se contar o número de resultados da consulta, sem realmente retorná-los:

```
( (Integer) session.createQuery("select count(*) from ....").iterate().next() ).intValue()
```

Para ordenar um resultado pelo tamanho de uma coleção, use a consulta 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 subseleções, pode-se colocar uma condição sobre tamanho de seleção na cláusula where da sua consulta:

```
from User usr where size(usr.messages)
>= 1
```

Se seu banco de dados não suporta subseleções, use a consulta a seguir:

```
select usr.id, usr.name
from User usr
    join usr.messages msg
group by usr.id, usr.name
having count(msg)
>= 1
```

Com essa solução não se pode retornar um `User` sem nenhuma menssagem, por causa da união inteira, 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 consulta:

```
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 coleções são paginàveis, 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 coleção podem ser ordenados ou agrupados usando um filtro de consulta:

```
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 coleção sem inicializá-la:

```
( (Integer) session.createQuery("select count(*) from ....").iterate().next() ).intValue();
```

16.17. Componentes

Os componentes podem ser usados de quase todas as formas que os tipos de valores simples são usados nas consultas HQL. Eles podem aparecer na cláusula `select`:

```
select p.name from Person p
```

```
select p.name.first from Person p
```

onde a propriedade do nome da Person é um componente. Os componentes também podem ser utilizados na cláusula `where`:

```
from Person p where p.name = :name
```

```
from Person p where p.name.first = :firstName
```

Os componentes também podem ser usados na cláusula `order by`:

```
from Person p order by p.name
```

```
from Person p order by p.name.first
```

Outro uso comum dos componentes é nos *row value constructors*.

16.18. Sintáxe do construtor de valores de linha

O HQL suporta o uso da sintáxe ANSI SQL `row value constructor`, algumas vezes chamado de sintáxe `tupla`, embora o banco de dados adjacente possa não suportar esta noção. Aqui nós geralmente nos referimos às comparações de valores múltiplos, tipicamente associada aos componentes. Considere uma entidade Person que define um componente de nome:

```
from Person p where p.name.first='John' and p.name.last='Jingleheimer-Schmidt'
```

Esta é uma sintáxe válida, embora um pouco verbosa. Seria ótimo tornar essa sintáxe um pouco mais concisa e utilizar a sintáxe `row value constructor`:

```
from Person p where p.name=('John', 'Jingleheimer-Schmidt')
```

Pode também ser útil especificar isto na cláusula `select`:

```
select p.name from Person p
```

Com o uso da sintáxe `row value constructor`, e que pode ser de benéfico, seria quando utilizar as subconsultas que precisem comparar com os valores múltiplos:

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

Ao decidir se você quer usar esta sintáxe ou não, deve-se considerar o fato de que a consulta será dependente da ordenação das sub-propriedades do componente no metadados.

Consultas por critérios

O Hibernate provê uma API de consulta por critério intuitiva e extensível.

17.1. Criando uma instância `Criteria`

A interface `org.hibernate.Criteria` representa a consulta ao invés de uma classe persistente particular. A sessão é uma fábrica para instâncias de `Criteria` .

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

17.2. Limitando o conjunto de resultados

Um critério individual de consulta é uma instância da interface `org.hibernate.criterion.Criterion`. A classe `org.hibernate.criterion.Restrictions` define os métodos da fábrica para obter certos tipos de `Criterion` pré fabricados.

```
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`). Um dos mais úteis permite especificar o SQL diretamente.

```
List cats = sess.createCriteria(Cat.class)
          .add(   Restrictions.sqlRestriction("lower({alias}.name)   like   lower(?)",   "Fritz
%", Hibernate.STRING) )
    .list();
```

O parâmetro `{alias}` será substituido pelo alias da entidade procurada.

Uma maneira alternativa de obter um critério é apartir de uma instância `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();
```

17.3. Ordenando resultados

Você poderá ordenar 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();
```

17.4. Associações

Através da navegação de associações usando `createCriteria()`, você pode especificar restrições por entidades relacionadas:

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

Note que o segundo `createCriteria()` retorna uma nova instância de `Criteria`, que refere aos elementos da coleção `kittens`.

A seguinte forma alternada é útil em certas circunstâncias:

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` não cria uma nova instância de `Criteria`.)

Note que as coleções de kittens mantidas pelas instâncias `Cat`, retornadas pelas duas consultas anteriores *não* são pré-filtradas pelo critério. Se você desejar recuperar somente os kittens que se encaixarem ao critérios, você deverá 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");
}
```

Você pode ainda manipular o conjunto do resultado usando a junção exterior restante:

```
               List cats = session.createCriteria( Cat.class )
                   .createAlias("mate", "mt", Criteria.LEFT_JOIN, Restrictions.like("mt.name",
 "good%") )
                      .addOrder(Order.asc("mt.age"))
```

```
                        .list();
```

Isto retornará todos os `Cats` com um mate (amigo) cujo nome inicia com "bom" ordenado pela idade de seu mate e todos os cats que não tem mates. Isto é útil quando houver necessidade de pedir ou limitar a prioridade do banco de dados em retornar conjuntos de resultado complexo/grande e remover muitas instâncias onde consultas múltiplas deveriam ter sido executadas e os resultados unidos pelo java em memória.

Sem este recurso, o primeiro de todos os cats sem um mate teria que ser carregado em uma consulta.

Uma segunda consulta teria que restaurar os cats com os mates cujos os nomes iniciem com "bom" selecionados pelas idades dos mates.

A terceira, em memória; as listas teriam que ser unidas manualmente.

17.5. Busca de associação dinâmica

Você deve especificar as semânticas de busca de associação 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();
```

This query will fetch both `mate` and `kittens` by outer join. See *Seção 21.1, “Estratégias de Busca ”* for more information.

17.6. Exemplos de consultas

A classe `org.hibernate.criterion.Example` permite que você construa um critério de consulta a partir de uma dada instância.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

Propriedades de versão, identificadores e associações são ignoradas. Por padrão, as propriedades de valor null são excluídas.

Você pode ajustar como o `Exemplo` 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 até usar os exemplos para colocar os critérios em objetos associados.

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

17.7. Projeções, agregações e agrupamento.

A classe `org.hibernate.criterion.Projections` é uma fábrica para instâncias de `Projection`. Você pode aplicar uma projeção à uma consulta, chamando o `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 há necessidade de um "agrupamento por" explícito em uma consulta por critério. Certos tipos de projeção são definidos para serem *projeções de agrupamento*, que também aparecem em uma cláusula `agrupamento por`SQL.

Um alias pode ser atribuído de forma opcional à uma projeção, assim o valor projetado pode ser referenciado em restrições ou ordenações. Aqui seguem duas formas diferentes para 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 instância de projeção à outra instância de `Projeção` em alias. Como um atalho, você poderá atribuir um alias quando adicionar a projeção à uma lista de 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 também pode usar um `Property.forName()` para 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();
```

17.8. Consultas e subconsultas desanexadas.

A classe `DetachedCriteria` deixa você criar uma consulta fora do escopo de uma sessão, e depois executá-la usando alguma `Session` arbitrária.

```
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();
```

Um `DetachedCriteria` também pode ser usado para expressar uma subconsulta. As instâncias de critérios, que envolvem subconsultas, podem ser obtidas através das `Subqueries` ou `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 as subconsultas correlacionadas 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();
```

17.9. Consultas por um identificador natural

Para a maioria das consultas, incluindo consultas de critérios, o cache de consulta não é muito eficiente, pois a invalidação do cache de consulta ocorre com muita frequência. No entanto, não há um tipo de consulta especial onde possamos otimizar um algoritmo de invalidação de cache: consultas realizadas por chaves naturais constantes. Em algumas aplicações, este tipo de consulta ocorre com freqüência. O API de critério provê provisão especial para este caso de uso.

Primeiro, você deve mapear a chave natural de sua entidade usando um `<natural-id>` e habilitar o uso de um 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
>
```

Note que esta funcionalidade não é proposta para o uso com entidades com chaves naturais *mutáveis*.

Uma vez que você tenha ativado o cache de consulta Hibernate, o `Restrictions.naturalId()` nos permite que utilizemos um algoritmo de cache mais eficiente.

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

SQL Nativo

Você também pode expressar consultas no dialeto SQL nativo de seu banco de dados. Isto é bastante útil para usar recursos específicos do banco de dados, assim como dicas de consultas ou a palavra chave em Oracle `CONNECT`. Ele também oferece um caminho de migração limpo de uma aplicação baseada em SQL/JDBC direta até o Hibernate.

O Hibernate3 permite que você especifique o SQL escrito à mão, incluindo procedimentos armazenados, para todas as operações de criar, atualizar, deletar e carregar.

18.1. Usando um `SQLQuery`

A execução de consultas SQL nativa é controlada através da interface `SQLQuery` que é obtido, chamando a `Session.createSQLQuery()`. As seções abaixo descrevem como usar este API para consultas.

18.1.1. Consultas Escalares

A consulta SQL mais básica é obter uma lista dos escalares (valores).

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

Eles irão retornar uma matriz de Lista de Objeto (Object[]) com valores escalares para cada coluna na tabela CATS. O Hibernate usará o ResultSetMetadata para deduzir a ordem atual e tipos de valores escalares retornados.

Para evitar o uso do `ResultSetMetadata` ou simplesmente para ser mais explícito em o quê é retornado, você poderá usar o `addScalar()`:

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

Esta consulta especificou:

- A string da consulta SQL
- as colunas e tipos para retornar

Este ainda irá retornar as matrizes de Objeto, mas desta vez ele não usará o `ResultSetMetdata`, ao invés disso, obterá explicitamente a coluna de ID, NOME e DATA DE NASCIMENTO como

respectivamente uma Longa, String e Curta a partir do conjunto de resultados adjacentes. Isto também significa que somente estas três colunas irão retornar, embora a consulta esteja utilizando `*` e possa retornar mais do que três colunas listadas.

É possível deixar de fora o tipo de informação para todos ou alguns dos escalares.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

Esta é a mesma consulta de antes, mas desta vez, o `ResultSetMetaData` é utilizado para decidir o tipo de NOME e DATA DE NASCIMENTO onde o tipo de ID é explicitamente especificado.

Como o java.sql.Types retornados do ResultSetMetadata é mapeado para os tipos Hibernate, ele é controlado pelo Dialeto. Se um tipo específico não é mapeado ou não resulta no tipo esperado, é possível padronizá-lo através de chamadas para `registerHibernateType` no Dialeto.

18.1.2. Consultas de Entidade

As consultas acima foram todas sobre o retorno de valores escalares, basicamente retornando os valores "não processados" do conjunto de resultados. A seguir, mostramos como obter objetos de entidade da consulta sql nativa através do `addEntity()`.

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

Esta consulta especificou:

- A string da consulta SQL
- A entidade retornada por uma consulta

Considerando que o Cat esteja mapeado como uma classe com colunas ID,NOME e DATA DE NASCIMENTO, as consultas acima irão devolver uma Lista onde cada elemento é uma entidade de Cat.

Se a entidade estiver mapeada com um `muitos-para-um` para outra entidade, requer-se que devolva também este ao desempenhar a consulta nativa, senão ocorrerá um erro de banco de dados específico "coluna não encontrada". As colunas adicionais serão automaticamente retornadas ao usar a anotação, mas preferimos ser explícitos como no seguinte exemplo para uma`muitos-para-um` para um `Dog`:

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

Isto irá permitir que o cat.getDog() funcione de forma apropriada

18.1.3. Manuseio de associações e coleções

É possível realizar a recuperação adiantada no `Dog` para evitar uma viagem extra por inicializar o proxy. Isto é feito através do método `addJoin()`que permite que você se una à associação ou coleção.

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d
 WHERE c.DOG_ID = d.D_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

Neste exemplo, a devolução do `Cat` terá sua propriedade `dog` totalmente inicializada sem nenhuma viagem extra ao banco de dados. Note que adicionamos um nome alias ("cat") para poder especificar o caminho da propriedade alvo na união. É possível fazer a mesma união para coleções, ex.: se ao invés disso, o `Cat` tivesse um-para-muitos para `Dog`.

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE
 c.ID = d.CAT_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

Neste estágio, estamos chegando no limite do que é possível fazer com as consultas nativas sem começar a destacar as colunas sql para torná-las útil no Hibernate. Os problemas começam a surgir quando se retorna entidades múltiplas do mesmo tipo ou quando o padrão de nomes de alias/coluna não são suficientes.

18.1.4. Retorno de entidades múltiplas

Até aqui, os nomes de colunas do conjunto de resultados são considerados como sendo os mesmos que os nomes de colunas especificados no documento de mapeamento. Isto pode ser problemático para as consultas SQL, que une tabelas múltiplas, uma vez que os mesmos nomes de colunas podem aparecer em mais de uma tabela.

É necessário uma injeção de alias de coluna na seguinte consulta (a qual é bem provável que falhe):

```
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 retornar duas instâncias Cat por linha: um cat e sua mãe. Isto irá falhar pois existe um conflito de nomes, são mapeados aos mesmos nomes de colunas e em alguns bancos de dados os aliases de colunas retornadas estarão, muito provavelmente, na forma

de "c.ID", "c.NOME", etc., os quais não são iguais às colunas especificadas no mapeamento ("ID" e "NOME").

A seguinte forma não é vulnerável à duplicação do nome 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 da consulta SQL, com espaço reservado para Hibernate para injetar aliases de coluna.
- as entidades retornadas pela consulta

A anotação {cat.*} e {mãe.*} usada acima, é um atalho para "todas as propriedades". De forma alternativa, você pode listar as colunas explicitamente, mas até neste caso nós deixamos o Hibernate injetar os aliases de coluna SQL para cada propriedade. O espaço reservado para um alias de coluna é simplesmente o nome de propriedade qualificado pelo alias de tabela. No seguinte exemplo, recuperamos os Cats e suas mães de uma tabela diferente (cat_log) para aquele declarado no metadado de mapeamentos. Note que podemos até usar os aliases de propriedade 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()
```

18.1.4.1. Alias e referências de propriedades

Para a maioria dos casos, necessita-se da injeção de alias acima. Para consultas relatadas aos mapeamentos mais complexos, como as propriedades compostas, discriminadores de herança, coleções, etc., você pode usar aliases específicos que permitem o Hibernate injetar os aliases apropriados.

As seguintes tabelas mostram as diferentes formas de usar uma injeção de alias. Por favor note que os nomes de alias no resultado são exemplos, cada alias terá um nome único e provavelmente diferente quando usado.

Tabela 18.1. Nomes de injeção de alias

Descrição	Sintáxe	Exemplo
Uma propriedade simples	`{[aliasname].[propertyname]`	`A_NAME as {item.name}`

Descrição	Sintáxe	Exemplo
Uma propriedade composta	`{[aliasname].[componentname].[propertyname]}`	`CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}`
Discriminador 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 elemento na coleção	`{[aliasname].element.[propertyname]}`	`NAME as {coll.element.name}`
Todas as propriedades de elemento na coleção	`{[aliasname].element.*}`	`{coll.element.*}`
Todas as propriedades da coleção	`{[aliasname].*}`	`{coll.*}`

18.1.5. Retorno de entidades não gerenciadas

É possível aplicar um ResultTransformer para consultas sql nativas, permitindo que o retorno de entidades não gerenciadas.

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

Esta consulta especificou:

- A string da consulta SQL
- um transformador de resultado

A consulta acima irá devolver uma lista de `CatDTO` que foi instanciada e injetada com valores dos comandos NAME e BIRTHDATE em suas propriedades correspondentes ou campos.

18.1.6. Manuseio de herança

As consultas sql nativas, as quais consultam entidades mapeadas como parte de uma herança, devem incluir todas as propriedades na classe base e todas as suas subclasses.

18.1.7. Parâmetros

Consultas sql Nativas suportam parâmetros posicionais assim como parâmetros nomeados:

```
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();
```

18.2. Consultas SQL Nomeadas

Named SQL queries can also be defined in the mapping document and called in exactly the same way as a named HQL query (see *Seção 11.4.1.7, "Externando consultas nomeadas"*). In this case, you do *not* need to call `addEntity()`.

Exemplo 18.1. Named sql query using the <sql-query> maping element

```
<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>
```

Exemplo 18.2. Execution of a named query

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

Os elementos `<return-join>` e `<load-collection>` são usados para unir associações e definir consultas que inicializam coleções,

Exemplo 18.3. Named sql query with association

```
<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},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

Uma consulta SQL nomeada pode devolver um valor escalar. Você deve declarar um alias de coluna e um tipo Hibernate usando o elemento `<return-scalar>`:

Exemplo 18.4. Named query returning a 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 externar as informações de mapeamento de conjunto de resultado em um elemento `<resultset>` tanto para reusá-los em diversas consultas nomeadas quanto através da API `setResultSetMapping()`.

Exemplo 18.5. <resultset> mapping used to externalize mapping information

```
<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},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
```

```
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

Você pode também, como forma alternativa, usar a informação de mapeamento de conjunto de resultado em seus arquivos hbm em código de java.

Exemplo 18.6. Programmatically specifying the result mapping information

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

So far we have only looked at externalizing SQL queries using Hibernate mapping files. The same concept is also available with anntations and is called named native queries. You can use `@NamedNativeQuery` (`@NamedNativeQueries`) in conjunction with `@SqlResultSetMapping` (`@SqlResultSetMappings`). Like `@NamedQuery`, `@NamedNativeQuery` and `@SqlResultSetMapping` can be defined at class level, but their scope is global to the application. Lets look at a view examples.

Exemplo 18.7, "Named SQL query using @NamedNativeQuery together with @SqlResultSetMapping" shows how a `resultSetMapping` parameter is defined in `@NamedNativeQuery`. It represents the name of a defined `@SqlResultSetMapping`. The resultset mapping declares the entities retrieved by this native query. Each field of the entity is bound to an SQL alias (or column name). All fields of the entity including the ones of subclasses and the foreign key columns of related entities have to be present in the SQL query. Field definitions are optional provided that they map to the same column name as the one declared on the class property. In the example 2 entities, `Night` and `Area`, are returned and each property is declared and associated to a column name, actually the column name retrieved by the query.

In *Exemplo 18.8, "Implicit result set mapping"* the result set mapping is implicit. We only describe the entity class of the result set mapping. The property / column mappings is done using the entity mapping values. In this case the model property is bound to the model_txt column.

Finally, if the association to a related entity involve a composite primary key, a `@FieldResult` element should be used for each foreign key column. The `@FieldResult` name is composed of the property name for the relationship, followed by a dot ("."), followed by the name or the field or property of the primary key. This can be seen in *Exemplo 18.9, "Using dot notation in @FieldResult for specifying associations "*.

Exemplo 18.7. Named SQL query using `@NamedNativeQuery` together with `@SqlResultSetMapping`

```
@NamedNativeQuery(name="night&area", query="select night.id nid, night.night_duration, "
```

```
    + " night.night_date, area.id aid, night.area_id, area.name "
    + "from Night night, Area area where night.area_id = area.id",
                    resultSetMapping="joinMapping")
@SqlResultSetMapping(name="joinMapping", entities={
    @EntityResult(entityClass=Night.class, fields = {
        @FieldResult(name="id", column="nid"),
        @FieldResult(name="duration", column="night_duration"),
        @FieldResult(name="date", column="night_date"),
        @FieldResult(name="area", column="area_id"),
        discriminatorColumn="disc"
    }),
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class, fields = {
        @FieldResult(name="id", column="aid"),
        @FieldResult(name="name", column="name")
    })
    }
)
```

Exemplo 18.8. Implicit result set mapping

```
@Entity
@SqlResultSetMapping(name="implicit",
                    entities=@EntityResult(entityClass=SpaceShip.class))
@NamedNativeQuery(name="implicitSample",
                  query="select * from SpaceShip",
                  resultSetMapping="implicit")
public class SpaceShip {
    private String name;
    private String model;
    private double speed;

    @Id
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Column(name="model_txt")
    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }
```

```
}
```

Exemplo 18.9. Using dot notation in @FieldResult for specifying associations

```
@Entity
@SqlResultSetMapping(name="compositekey",
        entities=@EntityResult(entityClass=SpaceShip.class,
            fields = {
                    @FieldResult(name="name", column = "name"),
                    @FieldResult(name="model", column = "model"),
                    @FieldResult(name="speed", column = "speed"),
                    @FieldResult(name="captain.firstname", column = "firstn"),
                    @FieldResult(name="captain.lastname", column = "lastn"),
                    @FieldResult(name="dimensions.length", column = "length"),
                    @FieldResult(name="dimensions.width", column = "width")
                    }),
        columns = { @ColumnResult(name = "surface"),
                    @ColumnResult(name = "volume") } )

@NamedNativeQuery(name="compositekey",
     query="select name, model, speed, lname as lastn, fname as firstn, length, width, length
 * width as surface from SpaceShip",
    resultSetMapping="compositekey")
} )
public class SpaceShip {
    private String name;
    private String model;
    private double speed;
    private Captain captain;
    private Dimensions dimensions;

    @Id
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @ManyToOne(fetch= FetchType.LAZY)
    @JoinColumns( {
            @JoinColumn(name="fname", referencedColumnName = "firstname"),
            @JoinColumn(name="lname", referencedColumnName = "lastname")
            } )
    public Captain getCaptain() {
        return captain;
    }

    public void setCaptain(Captain captain) {
        this.captain = captain;
    }

    public String getModel() {
        return model;
```

```
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }

    public Dimensions getDimensions() {
        return dimensions;
    }

    public void setDimensions(Dimensions dimensions) {
        this.dimensions = dimensions;
    }
}

@Entity
@IdClass(Identity.class)
public class Captain implements Serializable {
    private String firstname;
    private String lastname;

    @Id
    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    @Id
    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}
```

Dica

If you retrieve a single entity using the default mapping, you can specify the `resultClass` attribute instead of `resultSetMapping`:

```
@NamedNativeQuery(name="implicitSample",       query="select       *       from
 SpaceShip", resultClass=SpaceShip.class)
public class SpaceShip {
```

In some of your native queries, you'll have to return scalar values, for example when building report queries. You can map them in the `@SqlResultsetMapping` through `@ColumnResult`. You actually can even mix, entities and scalar returns in the same native query (this is probably not that common though).

Exemplo 18.10. Scalar values via `@ColumnResult`

```
@SqlResultSetMapping(name="scalar", columns=@ColumnResult(name="dimension"))
@NamedNativeQuery(name="scalar",     query="select     length*width     as     dimension     from
 SpaceShip", resultSetMapping="scalar")
```

An other query hint specific to native queries has been introduced: `org.hibernate.callable` which can be true or false depending on whether the query is a stored procedure or not.

18.2.1. Utilizando a propriedade retorno para especificar explicitamente os nomes de colunas/alias

Com a `<return-property>` você pode informar explicitamente, quais aliases de coluna utilizar, ao invés de usar a sintáxe `{}` para deixar o Hibernate injetar seus próprios aliases. Por exemplo:

```
<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 funciona com colunas múltiplas. Isto resolve a limitação com a sintáxe `{}` que não pode permitir controle granulado fino de muitas propriedades de colunas múltiplas.

```
<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>
```

Observe que neste exemplo nós usamos `<return-property>` combinado à síntáxe `{}` para injeção. Permite que os usuários escolham como eles querem se referir à coluna e às propriedades.

Se seu mapeamento possuir um discriminador, você deve usar `<return-discriminator>` para especificar a coluna do discriminador.

18.2.2. Usando procedimentos de armazenamento para consultas

O Hibernate 3 apresenta o suporte para consultas através de procedimentos e funções armazenadas. A maior parte da documentação a seguir, é equivalente para ambos. Os procedimentos e funções armazenados devem devolver um conjunto de resultados como primeiros parâmetros externos para poder trabalhar com o Hibernate. Um exemplo disto é a função armazenada em Oracle 9 e versões posteriores como se segue:

```
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 no Hibernate você vai precisar mapeá-lo através de uma consulta nomeada

```
<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>
```

Observe que os procedimentos armazenados somente devolvem escalares e entidades. O `<return-join>` e `<load-collection>` não são suportados.

18.2.2.1. Regras e limitações para utilizar procedimentos armazenados.

Para usar procedimentos armazenados com Hibernate, os procedimentos e funções precisam seguir a mesma regra. Caso não sigam estas regras, não poderão ser usados com o Hibernate. Se você ainda desejar usar estes procedimentos, terá que executá-los através da `session.connection()`. As regras são diferentes para cada banco de dados, uma vez que os fabricantes possuem procedimentos de semânticas/sintáxe armazenados.

Consultas de procedimento armazenado não podem ser paginados com o `setFirstResult()/setMaxResults()`.

O formulário de chamada recomedado é o padrão SQL92: `{ ? = call functionName(<parameters>) }` or `{ ? = call procedureName(<parameters>}`. A sintáxe de chamada nativa não é suportada.

As seguintes regras se aplicam para Oracle:

- A função deve retornar um conjunto de resultado. O primeiro parâmetro do procedimento deve ser um `OUT` que retorne um conjunto de resultado. Isto é feito usando o tipo `SYS_REFCURSOR` no Oracle 9 ou 10. No Oracle é necessário definir o tipo de `REF CURSOR`, veja a documentação do Oracle.

Para servidores Sybase ou MS SQL aplicam-se as seguintes regras:

- O procedimento deve retornar um conjunto de resultados. Observe que, como este servidor pode retornar múltiplos conjuntos de resultados e contas atualizadas, o Hibernate irá inteirar os resultados e pegar o primeiro resultado, o qual é o valor de retorno do conjunto de resultados. O resto será descartado.
- Se você habilitar `SET NOCOUNT ON` no seu procedimento, ele provavelmente será mais eficiente. Mas, isto não é obrigatório

18.3. SQL padronizado para criar, atualizar e deletar

Hibernate3 can use custom SQL for create, update, and delete operations. The SQL can be overridden at the statement level or inidividual column level. This section describes statement overrides. For columns, see *Seção 5.6, "Column transformers: read and write expressions"*. *Exemplo 18.11, "Custom CRUD via annotations"* shows how to define custom SQL operatons using annotations.

Exemplo 18.11. Custom CRUD via annotations

```
@Entity
@Table(name="CHAOS")
@SQLInsert( sql="INSERT INTO CHAOS(size, name, nickname, id) VALUES(?,upper(?),?,?)")
@SQLUpdate( sql="UPDATE CHAOS SET size = ?, name = upper(?), nickname = ? WHERE id = ?")
@SQLDelete( sql="DELETE CHAOS WHERE id = ?")
@SQLDeleteAll( sql="DELETE CHAOS")
@Loader(namedQuery = "chaos")
@NamedNativeQuery(name="chaos", query="select id, size, name, lower( nickname ) as nickname from
 CHAOS where id= ?", resultClass = Chaos.class)
public class Chaos {
    @Id
    private Long id;
    private Long size;
    private String name;
    private String nickname;
```

`@SQLInsert`, `@SQLUpdate`, `@SQLDelete`, `@SQLDeleteAll` respectively override the INSERT, UPDATE, DELETE, and DELETE all statement. The same can be achieved using Hibernate mapping files and the `<sql-insert>`, `<sql-update>` and `<sql-delete>` nodes. This can be seen in *Exemplo 18.12, "Custom CRUD XML"*.

Exemplo 18.12. Custom CRUD XML

```
<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>
```

If you expect to call a store procedure, be sure to set the `callable` attribute to `true`. In annotations as well as in xml.

To check that the execution happens correctly, Hibernate allows you to define one of those three strategies:

- none: no check is performed: the store procedure is expected to fail upon issues
- count: use of rowcount to check that the update is successful
- param: like COUNT but using an output parameter rather that the standard mechanism

To define the result check style, use the `check` parameter which is again available in annoations as well as in xml.

You can use the exact same set of annotations respectively xml nodes to override the collection related statements -see *Exemplo 18.13, "Overriding SQL statements for collections using annotations"*.

Exemplo 18.13. Overriding SQL statements for collections using annotations

```
@OneToMany
@JoinColumn(name="chaos_fk")
@SQLInsert( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = ? where id = ?")
@SQLDelete( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = null where id = ?")
private Set<CasimirParticle> particles = new HashSet<CasimirParticle>();
```

Dica

The parameter order is important and is defined by the order Hibernate handles properties. You can see the expected order by enabling debug logging for the `org.hibernate.persister.entity` level. With this level enabled Hibernate will print out the static SQL that is used to create, update, delete etc. entities. (To see the expected sequence, remember to not include your custom SQL through annotations or mapping files as that will override the Hibernate generated static sql)

Overriding SQL statements for secondary tables is also possible using `@org.hibernate.annotations.Table` and either (or all) attributes `sqlInsert`, `sqlUpdate`, `sqlDelete`:

Exemplo 18.14. Overriding SQL statements for secondary tables

```
@Entity
@SecondaryTables({
    @SecondaryTable(name = "`Cat nbr1`"),
    @SecondaryTable(name = "Cat2"})
@org.hibernate.annotations.Tables( {
    @Table(appliesTo = "Cat", comment = "My cat table" ),
    @Table(appliesTo = "Cat2", foreignKey = @ForeignKey(name="FK_CAT2_CAT"), fetch = FetchMode.SELECT,
        sqlInsert=@SQLInsert(sql="insert into Cat2(storyPart2, id) values(upper(?), ?)") )
} )
```

```
public class Cat implements Serializable {
```

The previous example also shows that you can give a comment to a given table (primary or secondary): This comment will be used for DDL generation.

Dica

The SQL is directly executed in your database, so you can use any dialect you like. This will, however, reduce the portability of your mapping if you use database specific SQL.

Last but not least, stored procedures are in most cases required to return the number of rows inserted, updated and deleted. Hibernate always registers the first statement parameter as a numeric output parameter for the CUD operations:

Exemplo 18.15. Stored procedures and their return value

```
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;
```

18.4. SQL padronizado para carga

You can also declare your own SQL (or HQL) queries for entity loading. As with inserts, updates, and deletes, this can be done at the individual column level as described in *Seção 5.6, "Column transformers: read and write expressions"* or at the statement level. Here is an example of a statement level override:

```
<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 instrução de consulta nomeada, como discutido anteriormente. Você pode referenciar esta consulta nomeada 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>
```

Este também funciona com procedimentos armazenados.

Você pode também definir uma consulta para carregar uma coleção:

```
<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 até definir um carregador de entidade que carregue uma coleção por busca de união:

```
<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>
```

The annotation equivalent `<loader>` is the @Loader annotation as seen in *Exemplo 18.11, "Custom CRUD via annotations"*.

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 ser habilitado ou não dentro de uma Sessão do Hibernate.

19.1. Filtros do Hibernate

O Hibernate3 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 à função "where" disponível para a classe e várias coleções. A não ser que essas condições de filtros possam ser parametrizadas. A aplicação pode, então decidir, 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 parâmetros dentro da aplicação.

Using annotatons filters are defined via `@org.hibernate.annotations.FilterDef` or `@org.hibernate.annotations.FilterDefs`. A filter definition has a `name()` and an array of parameters(). A parameter will allow you to adjust the behavior of the filter at runtime. Each parameter is defined by a `@ParamDef` which has a name and a type. You can also define a `defaultCondition()` parameter for a given `@FilterDef` to set the default condition to use when none are defined in each individual `@Filter`. `@FilterDef`(s) can be defined at the class or package level.

We now need to define the SQL filter clause applied to either the entity load or the collection load. `@Filter` is used and placed either on the entity or the collection element. The connection between `@FilterName` and `@Filter` is a matching name.

Exemplo 19.1. @FilterDef and @Filter annotations

```
@Entity
@FilterDef(name="minLength", parameters=@ParamDef( name="minLength", type="integer" ) )
@Filters( {
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length"),
    @Filter(name="minLength", condition=":minLength <= length")
} )
public class Forest { ... }
```

When the collection use an association table as a relational representation, you might want to apply the filter condition to the association table itself or to the target entity table. To apply the constraint on the target entity, use the regular `@Filter` annotation. However, if you want to target the association table, use the `@FilterJoinTable` annotation.

Exemplo 19.2. Using `@FilterJoinTable` for filterting on the association table

```
@OneToMany
    @JoinTable
    //filter on the target entity table
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length")
    //filter on the association table
    @FilterJoinTable(name="security", condition=":userlevel >= requredLevel")
    public Set<Forest> getForests() { ... }
```

Using Hibernate mapping files for defining filters the situtation is very similar. The filters must first be defined and then attached to the appropriate mapping elements. To define a filter, use the `<filter-def/>` element within a `<hibernate-mapping/>` element:

Exemplo 19.3. Defining a filter definition via `<filter-def>`

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

This filter can then be attached to a class or collection (or, to both or multiples of each at the same time):

Exemplo 19.4. Attaching a filter to a class or collection using `<filter>`

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>

    <set ...>
        <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
    </set>
</class>
```

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 sessão. 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 do método, comum à maioria das funções do Hibernate.

Um exemplo completo, usando dados temporais com um padrão de datas de registro efetivo:

```
<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 sessão antes de recuperar os dados dos empregados:

```
Session session = ...;
session.enableFilter("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 tenhamos mencionado 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 união externa (por HQL ou por busca de carga) seja cuidadoso quanto à direção da expressão de condição. É mais seguro configurá-lo para uma união externa esquerda. Coloque o parâmetro primeiro seguido pelo(s) nome(s) da coluna após o operador.

Após ser definido, o filtro deve ser anexado às entidades múltiplas e/ou coleções, cada uma com sua própria condição. Isto pode ser tedioso quando as condições se repetem. Assim, usando o `<filter-def/>` permite denifir uma condição padrão, tanto como uma função quanto CDATA:

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-def>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

Esta condição padrão será utilizada todas as vezes que um filtro for anexado a algo sem uma condição específica. Note que isto significa que você pode dar uma condição específica como parte de um anexo de filtro que substitua a condição padrão neste caso em particular.

Mapeamento XML

XML Mapping is an experimental feature in Hibernate 3.0 and is currently under active development.

20.1. Trabalhando com dados em XML

O Hibernate permite que se trabalhe com dados persistentes em XML quase da mesma maneira como você trabalha com POJOs persistentes. Uma árvore XML analisada, pode ser considerada 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, analisá-lo usando o dom4j, e escrever as alterações no banco de dados usando quaisquer operações básicas do Hibernate: `persist()`, `saveOrUpdate()`,`merge()`, `delete()`, `replicate()` (a mesclagem ainda não é suportada)

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.

20.1.1. 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
>
```

20.1.2. 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 gráfico de pares de nome/valor de propriedade ou `Maps` do Java. Os nomes de propriedades são somente construções lógicas que podem ser referenciadas em consultas HQL.

20.2. Mapeando metadados com XML

Muitos elementos do mapeamento do Hibernate aceitam a função `node`. Através dele, você pode especificar o nome de uma função ou elemento XML que contenha a propriedade ou os dados da entidade. O formato da função `node` deve ser o seguinte:

- `"element-name"`: mapeia para o elemento XML nomeado
- `"@attribute-name"`: mapeia para a função XML com determinado nome
- `"."`: mapeia para o elemento pai
- `"element-name/@attribute-name"`: mapeia para a função nomeada com o elemento nomeado

Para coleções e associações de valores simples, existe uma função adicional `embed-xml`. Se a função `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 as coleções simplesmente não irão aparecer.

Você precisa tomar cuidado para 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 incorporar a coleção de ids de contas, e não os dados de contas. Segue a abaixo a consulta HQL:

```
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 ajustar `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
>
```

20.3. Manipulando dados em XML

Vamos reler e atualizar documentos em XML em nossa aplicação. Nós fazemos isso obtendo uma sessão 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 de dados baseados em XML.

Aumentando o desempenho

21.1. Estratégias de Busca

Uma *estratégia de busca* é a estratégia que o Hibernate irá usar para recuperar objetos associados se a aplicação precisar navegar pela associação. Estratégias de Busca podem ser declaradas nos metadados de mapeamento O/R, ou sobrescritos por uma consulta HQL ou consulta com `Criteria`.

Hibernate3 define as seguintes estratégias de busca:

- *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 a busca lazy, especificando `lazy="false"`, esse segundo SELECT será executado apenas quando você acessar a associação.
- *Subselect fetching* - um segundo `SELECT` será usado para recuperar as coleções associadas de todas as entidades recuperadas em uma consulta ou busca anterior. A menos que você desabilite a busca lazy especificando `lazy="false"`, esse segundo SELECT será executado apenas quando você acessar a associação.
- *Batch fetching* - uma opção de otimização para selecionar a busca. O Hibernate recupera um lote de instâncias ou entidades usando um único `SELECT`, especificando uma lista de chaves primárias ou chaves externas.

O Hibernate distingue também entre:

- *Immediate fetching* - uma associação, coleção ou função é imediatamente recuperada, quando o proprietário for carregado.
- *Lazy collection fetching* - a coleção é recuperada 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 a partir do banco de dados quando necessário. O Hibernate tenta não buscar a coleção inteira dentro da memória a menos que seja absolutamente necessário. Isto é 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 único valor é recuperada quando a variável da instância é acessada. Comparada à busca proxy, esse método é menos preguiçoso

(lazy); a associação é buscada até mesmo quando somente o identificador é acessado. Ela 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 buscado 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 buscada e *como* ela é buscada. É importante que você não os confuda. Nós usamos `fetch` para ajustar o desempenho. Podemos usar `lazy` para definir um contrato para qual dado é sempre disponível em qualquer instância desconectada de uma classe particular.

21.1.1. 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.

Se você ajustar `hibernate. default_batch_fetch_size`, o Hibernate irá usar otimização de busca em lote para a busca preguiçosa. Essa otimização pode ser também habilitada em um nível mais fino.

Perceba que o 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` for fechada, a coleção não poderá carregar o seu estado. *O Hibernate não suporta inicialização preguiçosa para objetos desconectados*. Para consertar isso, é necessário mover o código que carrega a coleção para logo antes da transação ser submetida.

Alternativamente, nós podemos usar uma coleção ou associação não preguiçosa, especificando `lazy="false"` para o mapeamento da associação. Porém, é pretendido que a inicialização preguiçosa seja usada por quase todas as coleções e associações. Se você definir muitas associações não preguiçosas em seu modelo de objetos, o Hibernate irá precisar buscar no banco de dados inteiro da memória em cada transação.

Por outro lado, nós geralmente escolhemos a busca de união (que não é preguiçosa por natureza) ao invés do selecionar busca em uma transação particular. Nós agora veremos como customizar a estratégia de busca. No Hibernate3, os mecanismos para escolher a estratégia de busca são idênticos para as associações de valor único e para coleções.

21.1.2. Personalizando as estratégias de busca

O padrão selecionar busca, é extremamente vunerável aos problemas de seleção N+1, então habilitaremos a busca de união 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
- consultas por `Criteria`
- consultas HQL se a busca por `subselect` for usada

Independentemente da estratégia de busca que você usar, o gráfico não preguiçoso definido será certamente carregado na memória. Note que isso irá resultar em diversas seleções imediatas sendo usadas para rodar uma consulta HQL em particular.

Geralmente, 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 uma união externa. 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 consulta por `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.

21.1.3. Proxies de associação final único

A recuperação preguiçosa para coleções é implementada usando uma implementação própria do Hibernate para coleções persistentes. Porém, é necessário um mecanismo diferente para comportamento preguiçoso em associações de final único. A entidade alvo da associação precisa usar um proxy. O Hibernate implementa proxies para inicialização preguiçosa em objetos persistentes usando manipulação de bytecode, através da excelente biblioteca CGLIB.

Por padrão, o Hibernate3 gera proxies (na inicialização) para todas as classes persistentes que os usem para habilitar recuperação preguiçosa de associações `many-to-one` e `one-to-one`.

O arquivo de mapeamento deve declarar uma interface para usar como interface de proxy para aquela classe, com a função `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 estender esse comportamento para classes polimórficas. Por exemplo:

```
<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 instância 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!
    ....
}
```

E, segundo, 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 adjacente será do mesmo objeto:

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

E por terceiro, você não pode usar um proxy CGLIB em uma classe `final` ou com quaisquer métodos `final`.

Finalmente, se o seu objeto persistente adquirir qualquer recurso durante a instanciação (ex. 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 se dão devido à limitação originária do modelo de herança simples do Java. Se você quiser evitar esses problemas em suas classes persistentes você deve implementar uma interface que declare seus métodos comerciais. Você deve especificar essas interfaces no arquivo de mapeamento onde `CatImpl` implementa a interface `Cat` e `DomesticCatImpl` implementa a interface `DomesticCat`. Por exemplo:

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

Então, os proxies para instâncias de `Cat` e `DomesticCat` podem ser retornadas pelo `load()` ou `iterate()`.

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.createQuery("from CatImpl as cat where cat.name='fritz'").iterate();
Cat fritz = (Cat) iter.next();
```

Nota

`list()` normalmente retorna proxies.

Relacionamentos são também inicializados de forma preguiçosa. 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 inicializações de proxy imediatas.

21.1.4. Inicializando coleções e proxies

Será lançada uma `LazyInitializationException` se uma coleção não inicializada ou proxy for acessado fora do escopo da `Session`, isto é, quando a entidade que contém a coleção ou que possua a referência ao proxy estiver no estado desanexado.

Algumas vezes precisamos garantir que o proxy ou coleção é inicializado antes de 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()` favorecem a aplicação para trabalhar com coleções ou proxies inicializados de forma preguiçosa. O `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.

Uma outra opção é manter a `Session` aberta até que todas as coleções e os proxies necessários sejam carregados. Em algumas arquiteturas de aplicações, particularmente onde o código que acessa os dados usando Hibernate e o código que os usa, se encontram 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 uma aplicações web, um filtro servlet pode ser usado para fechar a `Session` somente no final da requisição do usuário, quando a renderização da view estiver completa (o modelo *Abrir Sessão em View*). Claro, que isto demanda uma exatidão no manuseio de exceções na infraestrutura de sua aplicação. É extremamente importante que a `Session` seja 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 "Abrir Sessão em 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 que é representada para um caso de uso particular. Geralmente, a aplicação chama `Hibernate.initialize()` para cada coleção que será usada pela camada web (essa chamada deve ocorrer antes da sessão ser fechada) ou retorna a

coleção usando uma consulta Hibernate com uma cláusula `FETCH` ou um `FetchMode.JOIN` na `Criteria`. Fica muito mais fácil se você adotar o modelo *Command* ao invés do *Session Facade*.

- Você também pode anexar um objeto previamente carregado em uma nova `Sessionmerge()` ou `lock()` antes de acessar coleções não inicializadas (ou outros proxies). O Hibernate não faz e certamente `não deve` fazer isso automaticamente, pois isso introduziria semântica em transações impromptu.

Às vezes você não quer inicializar uma coleção muito grande, mas precisa de algumas informações, como o mesmo tamanho, ou um subconjunto de seus dados.

Você pode usar um filtro de coleção para saber seu tamanho sem inicializá-la:

```
( (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();
```

21.1.5. Usando busca em lote

O Hibernate pode fazer uso eficiente de busca em lote, ou seja o Hibernate pode carregar diversos proxies não inicializados, se um proxy for acessado (ou coleções). A busca em lote é uma otimização da estratégia da busca de seleção lazy. Existem duas maneiras para você usar a 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` possui uma referência ao seu `owner`, que é da classe `Person`. A classe `Person` é mapeada com um proxy, `lazy="true"`. Se você interar 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; o padrão é 10, 10, 5.

Você também pode habilitar busca em lote de uma coleção. Por exemplo, se cada `Person` tem uma coleção preguiçosa de `Cats` e 10 persons estão já carregadas em uma `Session`, serão gerados 10 `SELECTs` ao se interar todas as persons, um para cada chamada de `getCats()`.

Se você habilitar 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 3, o Hibernate irá carregar 3, 3, 3, 1 coleções em 4 `SELECT`s. Novamente, o valor da função depende do número esperado de coleções não inicializadas em determinada `Session`.

A busca em lote de coleções é particularmente útil quando você tem uma árvore encadeada de ítens, 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.

21.1.6. Usando busca de subseleção

Se uma coleção ou proxy simples precisa ser recuperado, o Hibernate carrega todos eles rodando novamente a consulta original em uma subseleção. Isso funciona da mesma maneira que busca em lote, sem carregar tanto.

21.1.7. Perfis de Busca

Another way to affect the fetching strategy for loading associated objects is through something called a fetch profile, which is a named configuration associated with the `org.hibernate.SessionFactory` but enabled, by name, on the `org.hibernate.Session`. Once enabled on a `org.hibernate.Session`, the fetch profile will be in affect for that `org.hibernate.Session` until it is explicitly disabled.

So what does that mean? Well lets explain that by way of an example which show the different available approaches to configure a fetch profile:

Exemplo 21.1. Specifying a fetch profile using `@FetchProfile`

```
@Entity
@FetchProfile(name = "customer-with-orders", fetchOverrides = {

  @FetchProfile.FetchOverride(entity = Customer.class, association = "orders", mode = FetchMode.JOIN)
})
public class Customer {
   @Id
   @GeneratedValue
   private long id;

   private String name;

   private long customerNumber;
```

```
   @OneToMany
   private Set<Order> orders;

   // standard getter/setter
   ...
}
```

Exemplo 21.2. Specifying a fetch profile using `<fetch-profile>` outside `<class>` node

```
<hibernate-mapping>
    <class name="Customer">
        ...
        <set name="orders" inverse="true">
            <key column="cust_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>
    <class name="Order">
        ...
    </class>
    <fetch-profile name="customer-with-orders">
        <fetch entity="Customer" association="orders" style="join"/>
    </fetch-profile>
</hibernate-mapping>
```

Exemplo 21.3. Specifying a fetch profile using `<fetch-profile>` inside `<class>` node

```
<hibernate-mapping>
    <class name="Customer">
        ...
        <set name="orders" inverse="true">
            <key column="cust_id"/>
            <one-to-many class="Order"/>
        </set>
        <fetch-profile name="customer-with-orders">
            <fetch association="orders" style="join"/>
        </fetch-profile>
    </class>
    <class name="Order">
        ...
    </class>
</hibernate-mapping>
```

Now normally when you get a reference to a particular customer, that customer's set of orders will be lazy meaning we will not yet have loaded those orders from the database. Normally this is a good thing. Now lets say that you have a certain use case where it is more efficient to load

the customer and their orders together. One way certainly is to use "dynamic fetching" strategies via an HQL or criteria queries. But another option is to use a fetch profile to achieve that. The following code will load both the customer *and* their orders:

Exemplo 21.4. Activating a fetch profile for a given `Session`

```
Session session = ...;
session.enableFetchProfile( "customer-with-orders" );  // name matches from mapping
Customer customer = (Customer) session.get( Customer.class, customerId );
```

Nota

`@FetchProfile` definitions are global and it does not matter on which class you place them. You can place the `@FetchProfile` annotation either onto a class or package (package-info.java). In order to define multiple fetch profiles for the same class or package `@FetchProfiles` can be used.

Apenas os perfis de busca em estilo são suportados, mas planeja-se o suporte de estilos adicionais. Consulte *HHH-3414* [http://opensource.atlassian.com/projects/hibernate/browse/HHH-3414] para maiores detalhes.

21.1.8. Usando busca preguiçosa de propriedade

O Hibernate3 suporta a busca lazy de propriedades individuais. Essa técnica de otimização é também conhecida como *grupos de busca*. Veja que esta é mais uma característica de marketing já que na prática, é mais importante a 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 de propriedade lazy, é preciso ajustar a função `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 de propriedades lazy requer instrumentação de bytecode. Se suas classes persistentes não forem melhoradas, o Hibernate irá ignorar silenciosamente essa configuração e usará a busca imediata.

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>
```

Uma forma diferente de evitar leitura de coluna desnecessária, ao menos para transações de somente leitura, deve-se usar os recursos de projeção do HQL ou consultas por Critério. Isto evita a necessidade de processamento de bytecode em build-time e é certamente uma melhor solução.

Você pode forçar a busca antecipada comum de propriedades usando `buscar todas as propriedades` no HQL.

21.2. O Cachê de Segundo Nível

Uma `Session` do Hibernate é um cache de nível transacional de dados persistentes. É possível configurar um cluster ou um cache de nível JVM (nível `SessionFactory`) em uma estrutura classe por classe e coleção por coleção. Você pode até mesmo plugar em um cache em cluster. Tenha cuidado, pois os caches nunca sabem das mudanças feitas em armazenamento persistente por um outro aplicativo. No entanto, eles podem ser configurados para dados em cache vencido regularmente.

You have the option to tell Hibernate which caching implementation to use by specifying the name of a class that implements `org.hibernate.cache.CacheProvider` using the property `hibernate.cache.provider_class`. Hibernate is bundled with a number of built-in integrations with the open-source cache providers that are listed in *Tabela 21.1, “Provedores de Cache ”*. You can also implement your own and plug it in as outlined above. Note that versions prior to Hibernate 3.2 use EhCache as the default cache provider.

Tabela 21.1. Provedores de Cache

Cache	Classe de provedor	Tipo	Segurança de Cluster	Cache de Consulta Suportado
Hashtable (não recomendado para uso de produção)	`org.hibernate.cache.HashtableCacheProvider`	memória		yes
EHCache	`org.hibernate.cache.EhCacheProvider`	memória, disco		yes
OSCache	`org.hibernate.cache.OSCacheProvider`	memória, disco		yes
SwarmCache	`org.hibernate.cache.SwarmCacheProvider`	clustered (ip multicast)	sim (invalidação em cluster)	
JBoss Cache 1.x	`org.hibernate.cache.TreeCacheProvider`	clustered (ip multicast)	sim (replicação)	sim (solicitação de sync. de relógio)
JBoss Cache 2	`org.hibernate.cache.jbc.JBossCacheRegionFactory`	clustered (ip multicast)	sim (invalidação ou replicação)	sim (solicitação de sync. de relógio)

21.2.1. Mapeamento de Cache

As we have done in previous chapters we are looking at the two different possibiltites to configure caching. First configuration via annotations and then via Hibernate mapping files.

By default, entities are not part of the second level cache and we recommend you to stick to this setting. However, you can override this by setting the `shared-cache-mode` element in your `persistence.xml` file or by using the `javax.persistence.sharedCache.mode` property in your configuration. The following values are possible:

- `ENABLE_SELECTIVE` (Default and recommended value): entities are not cached unless explicitly marked as cacheable.
- `DISABLE_SELECTIVE`: entities are cached unless explicitly marked as not cacheable.
- `ALL`: all entities are always cached even if marked as non cacheable.
- `NONE`: no entity are cached even if marked as cacheable. This option can make sense to disable second-level cache altogether.

The cache concurrency strategy used by default can be set globaly via the `hibernate.cache.default_cache_concurrency_strategy` configuration property. The values for this property are:

- `read-only`
- `read-write`
- `nonstrict-read-write`
- `transactional`

Nota

It is recommended to define the cache concurrency strategy per entity rather than using a global one. Use the `@org.hibernate.annotations.Cache` annotation for that.

Exemplo 21.5. Definition of cache concurrency strategy via `@Cache`

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
```

Hibernate also let's you cache the content of a collection or the identifiers if the collection contains other entities. Use the `@Cache` annotation on the collection property.

Exemplo 21.6. Caching collections using annotations

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

Exemplo 21.7, "@Cache annotation with attributes" shows the `@org.hibernate.annotations.Cache` annotations with its attributes. It allows you to define the caching strategy and region of a given second level cache.

Exemplo 21.7. `@Cache` annotation with attributes

```
@Cache(
```

```
    CacheConcurrencyStrategy usage();          (1)
    String region() default "";                (2)
    String include() default "all";            (3)
)
```

1. usage: the given cache concurrency strategy (NONE, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL)
2. region (optional): the cache region (default to the fqcn of the class or the fq role name of the collection)
3. `include` (optional): all to include all properties, non-lazy to only include non lazy properties (default all).

Let's now take a look at Hibernate mapping files. There the `<cache>` element of a class or collection mapping is used to configure the second level cache. Looking at *Exemplo 21.8, "The Hibernate <cache> mapping element"* the parallels to anotations is obvious.

Exemplo 21.8. The Hibernate `<cache>` mapping element

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"   (1)
    region="RegionName"                                               (2)
    include="all|non-lazy"                                            (3)
/>
```

1. `uso` (solicitado) especifica a estratégia de cache: `transacional`, `leitura-escrita`, `leitura-escrita não estrito` ou `somente leitura`
2. `region` (opcional: padrão à classe ou nome papel da coleção): especifica o nome da região do cache de segundo nível
3. `include` (opcional: padrão para `all`) `non-lazy`: especifica que a propriedade da entidade mapeada com `lazy="true"` pode não estar em cache quando o nível da função busca lazy for habilitada

Alternatively to `<cache>`, you can use `<class-cache>` and `<collection-cache>` elements in `hibernate.cfg.xml`.

Let's now have a closer look at the different usage strategies

21.2.2. Estratégia: somente leitura

Se sua aplicação precisar ler mas nunca modificar instâncias de uma classe persistente, pode-se utilizar um cache de `read-only`. Esta é a estratégia de desempenho mais simples e melhor. É também perfeitamente seguro para uso em um cluster.

21.2.3. Estratégia: leitura/escrita

Se a aplicação precisar atualizar dados, um cache de `read-write` pode ser mais apropriado. Esta estratégia de cache nunca deve ser usada se solicitado um nível de isolação de transação serializável. Se o cache for usado em um ambiente JTA, você deve especificar a propriedade `hibernate.transaction.manager_lookup_class`, nomeando uma estratégia por obter o `TransactionManager` JTA. Em outros ambientes, você deve assegurar que a transação está completa quando a `Session.close()` ou `Session.disconnect()` for chamada. Se desejar utilizar esta estratégia em um cluster, você deve assegurar que a implementação de cache adjacente suporta o bloqueio. Os provedores de cache built-in *não* suportam o bloqueamento.

21.2.4. Estratégia: leitura/escrita não estrita

Se a aplicação somente precisa atualizar dados ocasionalmente (ou seja, se for extremamente improvável que as duas transações tentem atualizar o mesmo ítem simultaneamente) e não for requerido uma isolação de transação estrita, o uso deum cache de `nonstrict-read-write` pode ser mais apropriado. Se um cache é usado em ambiente JTA, você deverá especificar o `hibernate.transaction.manager_lookup_class`. Em outros ambientes, você deve assegurar que a transação está completa quando a `Session.close()` ou `Session.disconnect()` for chamada.

21.2.5. Estratégia: transacional

A estratégia de cache `transactional` provê suporte para provedores de cache transacional completo como o JBoss TreeCache. Tal cache, deve ser usado somente em um ambiente JTA e você deverá especificar o `hibernate.transaction.manager_lookup_class`.

21.2.6. Compatibilidade de Estratégia de Concorrência de Cache Provedor

Importante

Nenhum provedor de cache suporta todas as estratégias de concorrência de cache.

A seguinte tabela mostra qual provedor é compatível com qual estratégia de concorrência.

Tabela 21.2. Suporte de Estratégia de Concorrência de Cache

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (não recomendado para uso de produção)	yes	yes	yes	

Cache	read-only	nonstrict-read-write	read-write	transactional
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss Cache 1.x	yes			yes
JBoss Cache 2	yes			yes

21.3. Gerenciando os caches

Quando passar um objeto para `save()`, `update()` ou `saveOrUpdate()` e quando recuperar um objeto usando um `load()`, `get()`, `list()`, `iterate()` ou `scroll()`, este objeto será adicionado ao cache interno da `Session`.

Quando o `flush()` for subsequentemente chamado, o estado deste objeto será sincronizado com o banco de dados. Se você não desejar que esta sincronização aconteça ou se você estiver processando uma grande quantidade de objetos e precisar gerenciar a memória de forma eficiente, o método `evict()` pode ser usado para remover o objeto de suas coleções de cache de primeiro nível.

Exemplo 21.9. Explcitly evicting a cached instance from the first level cache using `Session.evict()`

```
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 oferece um métodocontains() para determinar se uma instância pertence ao cache de sessão.

Para despejar completamente todos os objetos do cache de Sessão, chame `Session.clear()`

Para o cache de segundo nível, existem métodos definidos na `SessionFactory` para despejar o estado de cache de uma instância, classe inteira, instância de coleção ou papel de coleção inteiro.

Exemplo 21.10. Second-level cache eviction via `SessionFactoty.evict()` and `SessionFacyory.evictCollection()`

```
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 e escreve itens ao cache de segundo nível.
- `CacheMode.GET`: itens de leitura do cache de segundo nível. Não escreve ao cache de segundo nível, exceto quando atualizar dados.
- `CacheMode.PUT`: escreve itens ao cache de segundo nível. Não lê a partir do cache de segundo nível.
- `CacheMode.REFRESH`: escreve itens ao cache de segundo nível, mas não lê a partir do cache de segundo nível. Passa o efeito de `hibernate.cache.use_minimal_puts`, forçando uma atualização do cache de segundo nível para que todos os itens leiam a partir do banco de dados.

Para navegar o conteúdo do segundo nível ou região de cache de consulta, use o`Statistics` API:

Exemplo 21.11. Browsing the second-level cache entries via the `Statistics` API

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

Você precisará habilitar estatísticas e, opcionalmente, forçar o Hibernate a manter as entradas de cache em um formato mais compreensível:

Exemplo 21.12. Enabling Hibernate statistics

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

21.4. O Cache de Consulta

O conjunto de resultado de consulta pode também estar em cache. Isto é útil, somente para consultas que são rodadas freqüentemente com os mesmos parâmetros.

21.4.1. Ativação do cache de consulta

A aplicação do cache nos resultados de consulta introduz alguns resultados referentes o seu processamento transacional normal de aplicações. Por exemplo, se você realizar o cache nos

resultados de uma consulta do Person Hibernate, você precisará acompanhar quando estes resultados deverão ser inválidos devido alterações salvas no Person. Tudo isto, acompanhado com o fato de que a maioria dos aplicativos não recebem benefício algum ao realizar o cache nos resultados da consulta, levando o Hibernate a desativar o cache de resultados de consulta por padrão. Para uso do cache de consulta, você primeiro precisa ativar o cache de consulta:

```
hibernate.cache.use_query_cache true
```

Esta configuração cria duas novas regiões de cache:

- `org.hibernate.cache.StandardQueryCache`, mantendo os resultados da consulta com cache.
- `org.hibernate.cache.UpdateTimestampsCache`, mantém os timestamps das atualizações mais recentes para tabelas consultáveis. Elas são usadas para validar os resultados uma vez que elas são servidas a partir do cache de consulta.

Importante

If you configure your underlying cache implementation to use expiry or timeouts is very important that the cache timeout of the underlying cache region for the UpdateTimestampsCache be set to a higher value than the timeouts of any of the query caches. In fact, we recommend that the the UpdateTimestampsCache region not be configured for expiry at all. Note, in particular, that an LRU cache expiry policy is never appropriate.

Conforme mencionado acima, a maioria das consultas não se beneficiam do cache ou de seus resultados. Portanto por padrão, as consultas individuais não estão em cache mesmo depois de ativar o cache de consulta. Para habilitar o caching de resultados, chame `org.hibernate.Query.setCacheable(true)`. Esta chamada permite que a consulta procure por resultados de caches existentes ou adicione seus resultados ao cache quando for executado.

Nota

O cache de consulta não realiza o cache ao estado de entidades atuais no cache, ele apenas realiza o cache nos valores identificadores e resultados do tipo de valor. Por esta razão, o cache de consulta deve sempre ser usado em conjunção com o cache de segundo nível para as entidades esperadas a sofrerem o cache como parte de um cache de resultado de consulta (apenas com o cache de coleção).

21.4.2. Regiões de cache de consulta

Se você solicitar um controle de granulado fino com políticas de validade do cache de consulta, você poderá especificar uma região de cache nomeada para uma consulta em 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 você quiser forçar um cache de consulta para uma atualização de sua região (independente de quaisquer resultados com cache encontrados nesta região), você poderá usar `org.hibernate.Query.setCacheMode(CacheMode.REFRESH)`. Juntamente com a região que você definiu para o cache gerado, o Hibernate seletivamente forçará os resultados com cache, naquela região particular a ser atualizada. Isto é particularmente útil em casos onde dados adjacentes podem ter sido atualizados através de um processo em separado , além de ser uma alternativa mais eficiente se aplicada ao despejo de uma região de cache através de `SessionFactory.evictQueries()`.

21.5. Entendendo o desempenho da Coleção

Nas seções anteriores nós descrevemos as coleções e seus aplicativos. Nesta seção nós exploraremos mais problemas em relação às coleções no período de execução.

21.5.1. Taxonomia

O Hibernate define três tipos básicos de coleções:

- Coleções de valores
- Associações um-para-muitos
- Associações muitos-para-muitos

A classificação distingue as diversas tabelas e relacionamento de chave externa, mas não nos diz tudo que precisamos saber sobre o modelo relacional. Para entender completamente a estrutura relacional e as características de desempenho, devemos também considerar a estrutura da chave primária que é usada pelo Hibernate para atualizar ou deletar linhas de coleções. Isto sugere a seguinte classificação:

- Coleções indexadas
- conjuntos

- Bags

Todas as coleções indexadas (mapas, listas, matrizes) possuem uma chave primária, que consiste em colunas `<key>` e `<index>`. Neste caso, as atualizações de coleção são geralmente muito eficientes. A chave primária pode ser indexada de forma eficiente e uma linha em particular pode ser localizada de forma eficiente quando o Hibernate tentar atualizar ou deletá-la.

Os conjuntos possuem uma chave primária que consiste em `<key>` e colunas de elemento. Isto pode ser menos eficiente para alguns tipos de elementos de coleções, especialmente elementos compostos ou textos grandes ou ainda campos binários. O banco de dados pode não ser capaz de indexar uma chave primária complexa de forma tão eficiente. Por um outro lado, para associações um-para-muitos ou muitos-para-muitos, especialmente no caso de identificadores sintáticos, é bem provável que seja tão eficiente quanto. Se você quiser que o `SchemaExport` crie para você uma chave primária de um `<set>` você deverá declarar todas as colunas como `not-null="true"`.

Os mapeamentos `<idbag>` definem uma chave substituta, para que elas sejam sempre muito eficientes ao atualizar. Na verdade, este é o melhor caso.

As Bags são os piores casos. Como uma bag permite duplicar valores de elementos e não possui coluna de índice, não se deve definir nenhuma chave primária. O Hibernate não tem como distinguir entre linhas duplicadas. O Hibernate resolve este problema, removendo completamente em um único `DELETE` e recria a coleção quando mudar. Isto pode ser bastante ineficiente.

Note que para uma associação um-para-muitos, a chave primária pode não ser a chave primária física da tabela do banco de dados, mas mesmo neste caso, a classificação acima é ainda útil. Isto reflete como o Hibernate "localiza" linhas individuais da coleção.

21.5.2. Listas, mapas, bags de id e conjuntos são coleções mais eficientes para atualizar

A partir da discussão acima, deve ficar claro que as coleções indexadas e conjuntos (geralmente) permitem uma operação mais eficente em termos de adição, remoção e atualização de elementos.

Existe ainda, mais uma vantagem, das coleções indexadas sob conjuntos para associações muitos-para-muitos. Por causa da estrutura de um `Set`, o Hibernate nunca utiliza o comando `UPDATE` em uma linha quando um elemento é "modificado". As mudanças para o `Conjunto` funcionam sempre através do comando `INSERT` e `DELETE` de linhas individuais. Novamente, esta consideração não se aplica às associações um para muitos.

Após observar que as matrizes não podem ser preguiçosas, nós concluimos que as listas, mapas e bags de id são tipos de coleções com maior desempenho (não inverso), com conjuntos que não ficam atrás. Espera-se que os conjuntos sejam um tipo mais comum de coleção nas aplicações Hibernate. Isto porque as semânticas "conjunto" são mais naturais em modelos relacionais.

No entanto, em modelos de domínio de Hibernate bem criados, geralmente vemos que a maioria das coleções são de fato, associações um-para-muitos com `inverse="true"`. Para estas

associações, a atualização é manipulada pelo lado muitos-para-um de uma associação e portanto considerações de desempenho de atualização de coleção simplesmente não se aplicam a este caso.

21.5.3. As Bags e listas são as coleções de inversão mais eficientes.

Existe um caso em particular no qual as bags (e também as listas) possuem um desempenho muito maior do que conjuntos. Para uma coleção com `inverse="true"`, o idioma de relacionamento um-para-um bidirecional padrão, por exemplo, podemos adicionar elementos a uma bag ou uma lista sem precisar inicializar (buscar) os elementos da bag. Isto acontece porque a `Collection.add()` ou `Collection.addAll()` deve sempre retornar verdadeira para uma bag ou `List`. Isto pode fazer que o código comum seguinte seja 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();
```

21.5.4. Deletar uma vez

Às vezes, deletar elementos de coleção um por um pode ser extremamente ineficiente. O Hibernate não é completamente burro, portanto ele sabe que não deve fazer isso no caso de uma coleção que tenha sido esvaziada recentemente (se você chamou `list.clear()`, por exemplo). Neste caso, o Hibernate irá editar um único `DELETE`.

Vamos supor que tenha adicionado um elemento único à uma coleção de tamanho vinte e então remove dois elementos. O Hibernate irá editar uma instrução `INSERT` e duas instruções `DELETE`, a não ser que a coleção seja uma bag. Isto é certamente desejável.

No entanto, suponha que removamos dezoito elementos, deixando dois e então adicionando três novos elementos. Existem duas formas possíveis de se proceder:

- delete dezoito linhas uma por uma e então insira três linhas
- remova toda a coleção em um SQL `DELETE` e insira todos os cinco elementos atuais, um por um

O Hibernate não sabe que a segunda opção é provavelmente mais rápida neste caso. O Hibernate não deseha saber a opção, uma vez que tal comportamento deve confundir os triggers do banco de dados, etc.

Felizmente, você pode forçar este comportamento (ou seja, uma segunda estratégia) a qualquer momento, descartando (ou seja, desreferenciando) a coleção original e retornando uma coleção recentemente instanciada com todos os elementos atuais.

É claro que, deletar somente uma vez, não se aplica às coleções mapeadas `inverse="true"`.

21.6. Monitorando desempenho

A otimização não é muito usada sem o monitoramento e acesso ao número de desempenho. O Hibernate oferece uma grande variedade de números sobre suas operações internas. Estatísticas em Hibernate estão disponíveis através do `SessionFactory`.

21.6.1. Monitorando uma SessionFactory

Você poderá acessar as métricas da `SessionFactory` de duas formas. Sua primeira opção é chamar a `sessionFactory.getStatistics()` e ler ou dispôr as `Estatísticas` você mesmo.

O Hibernate também usa o JMX para publicar métricas se você habilitar o MBean de `StatisticsService`. Você deve habiliar um MBean único para todas as suas `SessionFactory` ou uma por factory. Veja o seguinte código para exemplos de configurações minimalísticos:

```
// 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
```

Você pode (des)ativar o monitoramento para uma `SessionFactory`:

- no tempo de configuração, ajuste `hibernate.generate_statistics` para `falso`
- em tempo de espera: `sf.getStatistics().setStatisticsEnabled(true)` ou `hibernateStatsBean.setStatisticsEnabled(true)`

As estatísticas podem ser reajsutadas de forma programática, usando o método `clear()`. Um resumo pode ser enviado para o usuário (nível de info) usando o método `logSummary()`.

21.6.2. Métricas

O Hibernate oferece um número de métricas, desde informações bem básicas até especializadas, somente relevantes a certos cenários. Todos os contadores disponíveis estão descritos na API da interface `Statistics`, em três categorias:

- As métricas relacionadas ao uso da `Sessão`, tal como um número de sessões em aberto, conexões JDBC recuperadas, etc.
- As métricas relacionadas às entidades, coleções, consultas e caches como um todo (mais conhecido como métricas globais).
- Métricas detalhadas relacionadas à uma entidade em particular, coleção, consulta ou região de cache.

Por exemplo, você pode verificar a coincidência de um cache, perder e colocar a relação entre as entidades, colações e consultas e tempo médio que uma consulta precisa. Esteja ciente de que o número de milisegundos é sujeito a aproximação em Java. O Hibernate é preso à precisão do JVM, em algumas plataformas a precisão chega a ser de 10 segundos.

Os Getters simples são usados para acessar métricas globais (ou seja, não presos à uma entidade em particular, coleção, região de cache, etc.) Você pode acessar as métricas de uma entidade em particular, coleção ou região de cache através de seu nome e através de sua representação de HQL ou SQL para consultas. Por favor consulte a Javadoc API `Statistics`, `EntityStatistics`, `CollectionStatistics`, `SecondLevelCacheStatistics`, e `QueryStatistics` para maiores informações. O seguinte código mostra 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 em todas as entidades, coleções, consultas e caches regionais, você poderá recuperar os nomes de lista de entidades, coleções, consultas e caches regionais com os seguintes métodos: `getQueries()`, `getEntityNames()`, `getCollectionRoleNames()`, e `getSecondLevelCacheRegionNames()`.

Guia de Toolset

É possível realizar uma engenharia de roundtrip com o Hibernate, usando um conjunto de plug-ins de Eclipse, ferramentas de linha de comando, assim como tarefas Ant.

As *Ferramentas do Hibernate* atualmente incluem os plug-ins para o IDE de Eclipse assim como as tarefas para reverter a engenharia dos bancos de dados existentes:

- *Editor de Mapeamento:* um editor para mapeamento de arquivos XML do Hibernate, suportando a auto complexão e destaque de sintáxe. Ele também suporta a auto complexão da semântica para nomes de classes e nomes de propriedade/campo, fazendo com que ele seja mais versátil do que um editor XML normal.
- *Console:* o console é uma nova visão em Eclipse. Além disso, para uma visão geral de árvore de suas configurações de console, você também pode obter uma visão interativa de suas classes persistentes e seus relacionamentos. O console permite que você execute as consultas HQL junto ao banco de dados e navegue o resultado diretamente em Eclipse.
- *Assistentes de Desenvolvimento:* são oferecidos diversos assistentes com as ferramentas de Eclipse do Hibernate. Você pode usar o assistente para gerar rapidamente arquivos de configuração do Hibernate (cfg.xml), ou você pode também reverter completamente o engenheiro, um esquema de banco de dados existente, para arquivos de fonte POJO e arquivos de mapeamento do Hibernate. O assistente de engenharia reversa suporta modelos padronizáveis.
-

Por favor, consulte o pacote *Ferramentas do Hibernate* e suas documentações para maiores informações.

No entanto, o pacote principal do Hibernate vem em lote com uma ferramenta integrada: *SchemaExport* aka `hbm2ddl`. Ele pode também ser usado dentro do Hibernate.

22.1. Geração de esquema automático

O DDL pode ser gerado a partir dos arquivos de mapeamento através dos utilitários do Hibernate. O esquema gerado inclui as restrições de integridade referencial, primária e chave estrangeira, para entidade e tabela de coleção. Tabelas e seqüência são também criadas por geradores de identificador mapeado.

Você *deve* especificar um SQL `Dialect` através da propriedade `hibernate.dialect` ao usar esta ferramenta, uma vez que o DDL é um fabricante bastante específico.

Primeiro, padronize seus arquivos de mapeamento para melhorar o esquema gerado. A próxima seção cobrirá a personalização do esquema.

22.1.1. Padronizando o esquema

Muitos elementos de mapeamento do Hibernate definem funções opcionais nomeadas `length`, `precision` e `scale`. Você deve ajustar o length, precision e scale de uma coluna com esta função.

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

Algumas tags aceitam uma função `not-null` para gerar uma restrição `NOT NULL`nas colunas de tabela e uma função `unique` para gerar uma restrição `UNIQUE` em colunas de tabela.

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

Uma função `unique-key` pode ser usada para agrupar colunas em uma restrição de chave única. Atualmente, o valor específico da função `unique-key` *não é* usada para nomear a restrição 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"/>
```

Uma função `index` especifica o nome de um indexe que será criado, usando a coluna ou colunas mapeada(s). As colunas múltiplas podem ser agrupadas no mesmo indexe, simplesmente especificando o mesmo nome de índice.

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

Uma função `foreign-key` pode ser usada para sobrescrever o nome de qualquer restrição de chave exterior gerada.

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

Muitos elementos de mapeamento também aceitam um elemento filho `<column>`. Isto é particularmente útil para mapeamento de tipos multi-colunas:

```
<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
>
```

A função `default` deixa você especificar um valor padrão para uma coluna. Você deve atribuir o mesmo valor à propriedade mapeada antes de salvar uma nova instância 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
>
```

A função `sql-type` permite que o usuário sobrescreva o mapeamento padrão de um tipo de Hibernate para um tipo de dados SQL.

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property
>
```

A função `check` permite que você especifique uma restrição de verificação.

```
<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
>
```

A seguinte tabela resume estes atributos opcionais.

Tabela 22.1. Sumário

Função	Valores	Interpretação
`length`	número	comprimento da coluna
`precision`	número	precisão da coluna decimal
`scale`	número	escaça de coluna decimal
`not-null`	`true|false`	especifica que a coluna deveria ser não anulável
`unique`	`true|false`	especifica que a coluna deveria ter uma restrição única
`index`	`index_name`	especifica o nome de um índice (multi-coluna)
`unique-key`	`unique_key_name`	especifica o nome de uma restrição única de coluna múltipla
`foreign-key`	`foreign_key_name`	especifica o nome da restrição de chave estrangeira gerada para uma associação, por um elemento de mapeamento `<one-to-one>`, `<many-to-one>`, `<key>`, ou `<many-to-many>`. Note que os lados `inverse="true"` não serão considerados pelo `SchemaExport`.
`sql-type`	`SQL column type`	sobrescreve o tipo de coluna padrão (função do elemento `<column>`somente)
`default`	Expressão SQL	especifica um valor padrão para a coluna
`check`	Expressão SQL	cria uma restrição de verificação de SQL tanto na coluna quanto na tabela

O elemento `<comment>` permite que você especifique comentários para esquema 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 uma instrução `comment on table` ou `comment on column` no DDL gerado, onde é suportado.

22.1.2. Executando a ferramenta

A ferramenta `SchemaExport` escreve um script DDL para padronizar e/ou para executar as instruções DDL.

A seguinte tabela exibe as opções de linha de comando do `SchemaExport`

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaExport` *options mapping_files*

Tabela 22.2. `SchemaExport` Opções de Linha de Comando

Opção	Descrição
`--quiet`	não saia do script para stdout
`--drop`	somente suspenda as tabelas
`--create`	somente crie tabelas
`--text`	não exporte para o banco de dados
`--output=my_schema.ddl`	saia do script ddl para um arquivo
`--naming=eg.MyNamingStrategy`	seleciona um `NamingStrategy`
`--config=hibernate.cfg.xml`	leia a configuração do Hibernate a partir do arquivo XML
`--properties=hibernate.properties`	leia propriedades de banco de dados a partir dos arquivos
`--format`	formatar bem o SQL gerado no script
`--delimiter=;`	ajustar e finalizar o delimitador de linha para o script

Você pode até mesmo incorporar o `SchemaExport` em sua aplicação:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

22.1.3. Propriedades

As Propriedades do Banco de Daods podem ser especificadas:

- Como Propriedades de sistema com `-D`*<property>*
- em `hibernate.properties`
- em um arquivo de propriedades nomeadas com `--properties`

As propriedades necessárias são:

Tabela 22.3. SchemaExport Connection Properties

Nome de Propriedade	Descrição
`hibernate.connection.driver_class`	classe de driver jdbc
`hibernate.connection.url`	jdbc url
`hibernate.connection.username`	usuário de banco de dados
`hibernate.connection.password`	senha do usuário
`hibernate.dialect`	dialeto

22.1.4. Usando o Ant

Você pode chamar o `SchemaExport` a partir de seu script de construção do 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
>
```

22.1.5. Atualizações de esquema incremental

A ferramenta `SchemaUpdate` irá atualizar um esquema existente com mudanças "incrementais". Observe que `SchemaUpdate` depende totalmente da API de metadados JDBC, portanto não irá funcionar com todos os driver JDBC.

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaUpdate` *options mapping_files*

Tabela 22.4. `SchemaUpdate`Opções de Linha de Comando

Opção	Descrição
`--quiet`	não saia do script para stdout
`--text`	não exporte o script ao banco de dados
`--naming=eg.MyNamingStrategy`	seleciona um `NamingStrategy`

Opção	Descrição
`--properties=hibernate.properties`	leia propriedades de banco de dados a partir dos arquivos
`--config=hibernate.cfg.xml`	especifique um arquivo `.cfg.xml`

Você pode incorporar o `SchemaUpdate` em sua aplicação:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

22.1.6. Utilizando Ant para atualizações de esquema incremental

Você pode chamar `SchemaUpdate` a partir do 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
>
```

22.1.7. Validação de esquema

A ferramenta `SchemaValidator` irá confirmar que o esquema de banco de dados existente "combina" com seus documentos de mapeamento. Observe que o `SchemaValidator` depende totalmente da API de metadados JDBC, portanto ele não funcionará com todos os drivers JDBC. Esta ferramenta é extremamente útil para teste.

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaValidator` *options mapping_files*

A seguinte tabela exibe as opções de linha de comando do `SchemaValidator`:

Tabela 22.5. `SchemaValidator` Opções de Linha de Comando

Opção	Descrição
`--naming=eg.MyNamingStrategy`	seleciona um `NamingStrategy`

Opção	Descrição
`--properties=hibernate.properties`	leia propriedades de banco de dados a partir dos arquivos
`--config=hibernate.cfg.xml`	especifique um arquivo `.cfg.xml`

Você pode incorporar o `SchemaValidator` em sua aplicação:

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

22.1.8. Utilizando Ant para validação de esquema

Você pode chamar o `SchemaValidator` a partir do 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>
    </schemavalidator>
</target
>
```

Additional modules

Hibernate Core also offers integration with some external modules/projects. This includes Hibernate Validator the reference implementation of Bean Validation (JSR 303) and Hibernate Search.

23.1. Bean Validation

Bean Validation standardizes how to define and declare domain model level constraints. You can, for example, express that a property should never be null, that the account balance should be strictly positive, etc. These domain model constraints are declared in the bean itself by annotating its properties. Bean Validation can then read them and check for constraint violations. The validation mechanism can be executed in different layers in your application without having to duplicate any of these rules (presentation layer, data access layer). Following the DRY principle, Bean Validation and its reference implementation Hibernate Validator has been designed for that purpose.

The integration between Hibernate and Bean Validation works at two levels. First, it is able to check in-memory instances of a class for constraint violations. Second, it can apply the constraints to the Hibernate metamodel and incorporate them into the generated database schema.

Each constraint annotation is associated to a validator implementation responsible for checking the constraint on the entity instance. A validator can also (optionally) apply the constraint to the Hibernate metamodel, allowing Hibernate to generate DDL that expresses the constraint. With the appropriate event listener, you can execute the checking operation on inserts, updates and deletes done by Hibernate.

When checking instances at runtime, Hibernate Validator returns information about constraint violations in a set of `ConstraintViolation`s. Among other information, the `ConstraintViolation` contains an error description message that can embed the parameter values bundle with the annotation (eg. size limit), and message strings that may be externalized to a `ResourceBundle`.

23.1.1. Adding Bean Validation

To enable Hibernate's Bean Validation integration, simply add a Bean Validation provider (preferably Hibernate Validation 4) on your classpath.

23.1.2. Configuration

By default, no configuration is necessary.

The `Default` group is validated on entity insert and update and the database model is updated accordingly based on the `Default` group as well.

You can customize the Bean Validation integration by setting the validation mode. Use the `javax.persistence.validation.mode` property and set it up for example in your `persistence.xml` file or your `hibernate.cfg.xml` file. Several options are possible:

- `auto` (default): enable integration between Bean Validation and Hibernate (callback and ddl generation) only if Bean Validation is present in the classpath.
- `none`: disable all integration between Bean Validation and Hibernate
- `callback`: only validate entities when they are either inserted, updated or deleted. An exception is raised if no Bean Validation provider is present in the classpath.
- `ddl`: only apply constraints to the database schema when generated by Hibernate. An exception is raised if no Bean Validation provider is present in the classpath. This value is not defined by the Java Persistence spec and is specific to Hibernate.

Nota

You can use both `callback` and `ddl` together by setting the property to `callback, dll`

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.mode"
                value="callback, ddl"/>
    </properties>
  </persistence-unit>
</persistence>
```

This is equivalent to `auto` except that if no Bean Validation provider is present, an exception is raised.

If you want to validate different groups during insertion, update and deletion, use:

- `javax.persistence.validation.group.pre-persist`: groups validated when an entity is about to be persisted (default to `Default`)
- `javax.persistence.validation.group.pre-update`: groups validated when an entity is about to be updated (default to `Default`)
- `javax.persistence.validation.group.pre-remove`: groups validated when an entity is about to be deleted (default to no group)
- `org.hibernate.validator.group.ddl`: groups considered when applying constraints on the database schema (default to `Default`)

Each property accepts the fully qualified class names of the groups validated separated by a comma (,)

Exemplo 23.1. Using custom groups for validation

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.group.pre-update"
                value="javax.validation.group.Default, com.acme.group.Strict"/>
      <property name="javax.persistence.validation.group.pre-remove"
                value="com.acme.group.OnDelete"/>
      <property name="org.hibernate.validator.group.ddl"
                value="com.acme.group.DDL"/>
    </properties>
  </persistence-unit>
</persistence>
```

Nota

You can set these properties in `hibernate.cfg.xml`, `hibernate.properties` or programmatically.

23.1.3. Catching violations

If an entity is found to be invalid, the list of constraint violations is propagated by the `ConstraintViolationException` which exposes the set of `ConstraintViolations`.

This exception is wrapped in a `RollbackException` when the violation happens at commit time. Otherwise the `ConstraintViolationException` is returned (for example when calling `flush()`. Note that generally, catchable violations are validated at a higher level (for example in Seam / JSF 2 via the JSF - Bean Validation integration or in your business layer by explicitly calling Bean Validation).

An application code will rarely be looking for a `ConstraintViolationException` raised by Hibernate. This exception should be treated as fatal and the persistence context should be discarded (`EntityManager` or `Session`).

23.1.4. Database schema

Hibernate uses Bean Validation constraints to generate an accurate database schema:

- `@NotNull` leads to a not null column (unless it conflicts with components or table inheritance)
- `@Size.max` leads to a `varchar(max)` definition for Strings

- `@Min`, `@Max` lead to column checks (like `value <= max`)
- `@Digits` leads to the definition of precision and scale (ever wondered which is which? It's easy now with `@Digits` :))

These constraints can be declared directly on the entity properties or indirectly by using constraint composition.

For more information check the Hibernate Validator *reference documentation* [http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/].

23.2. Hibernate Search

23.2.1. Description

Full text search engines like Apache Lucene™ are a very powerful technology to bring free text/efficient queries to applications. If suffers several mismatches when dealing with a object domain model (keeping the index up to date, mismatch between the index structure and the domain model, querying mismatch...) Hibernate Search indexes your domain model thanks to a few annotations, takes care of the database / index synchronization and brings you back regular managed objects from free text queries. Hibernate Search is using *Apache Lucene* [http://lucene.apache.org] under the cover.

23.2.2. Integration with Hibernate Annotations

Hibernate Search integrates with Hibernate Core transparently provided that the Hibernate Search jar is present on the classpath. If you do not wish to automatically register Hibernate Search event listeners, you can set `hibernate.search.autoregister_listeners` to false. Such a need is very uncommon and not recommended.

Check the Hibernate Search *reference documentation* [http://docs.jboss.org/hibernate/stable/search/reference/en-US/html/] for more information.

Exemplo: Pai/Filho

Uma das primeiras coisas que um usuário tenta fazer com o Hibernate é modelar um tipo de relacionamento Pai/Filho. Existem duas abordagens diferentes para isto. Por diversas razões diferentes, a abordagem mais conveniente, especialmente para novos usuários, é modelar ambos os `Parent` e `Child` como classes de entidade com uma associação `<one-to-many>` a partir do `Parent` para o `Child`. A abordagem alternativa é declarar o `Child` como um `<composite-element>`. As semânticas padrões da associação um para muitos (no Hibernate), são muito menos parecidas com as semânticas comuns de um relacionamento pai/filho do que aqueles de um mapeamento de elemento de composição. Explicaremos como utilizar uma *associação bidirecional um para muitos com cascatas* para modelar um relacionamento pai/filho de forma eficiente e elegante.

24.1. Uma nota sobre as coleções

As coleções do Hibernate são consideradas uma parte lógica de suas próprias entidades, nunca das entidades contidas. Saiba que esta é uma distinção que possui as seguintes conseqüências:

- Quando removemos ou adicionamos um objeto da/na coleção, o número da versão do proprietário da coleção é incrementado.
- Se um objeto removido de uma coleção for uma instância de um tipo de valor (ex.: um elemento de composição), este objeto irá parar de ser persistente e seu estado será completamente removido do banco de dados. Da mesma forma, ao adicionar uma instância de tipo de valor à coleção, causará ao estado uma persistência imediata.
- Por outro lado, se uma entidade é removida de uma coleção (uma associação um-para-muitos ou muitos-para-muitos), ela não será deletada por padrão. Este comportamento é completamente consistente, uma mudança para o estado interno de uma outra entidade não deve fazer com que a entidade associada desapareça. Da mesma forma, ao adicionar uma entidade à coleção, não faz com que a entidade se torne persistente, por padrão.

A adição de uma entidade à coleção, por padrão, meramente cria um link entre as duas entidades. A remoção da entidade, removerá o link. Isto é muito apropriado para alguns tipos de casos. No entanto, não é apropriado o caso de um relacionamento pai/filho. Neste caso, a vida do filho está vinculada ao ciclo de vida do pai.

24.2. Bidirecional um-para-muitos

Suponha que começamos uma associação `<one-to-many>` simples de `Parent` para `Child`.

```
<set name="children">
    <key column="parent_id"/>
```

```
    <one-to-many class="Child"/>
</set
>
```

Se fossemos executar o seguinte código:

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

O Hibernate editaria duas instruções SQL

- um `INSERT` para criar um registro para `c`
- um `UPDATE` para criar um link de `p` para `c`

Isto não é somente ineficiente como também viola qualquer restrição `NOT NULL` na coluna `parent_id`. Nós podemos concertar a violação da restrição de nulabilidade, especificando um `not-null="true"` no mapeamento da coleção:

```
<set name="children">
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set
>
```

No entanto, esta não é uma solução recomendada.

As causas subjacentes deste comportamento é que o link (a chave exterior `parent_id`) de `p` para `c` não é considerada parte do estado do objeto `Child` e por isso não é criada no `INSERT`. Então a solução é fazer uma parte de link do mapeamento do `Child`.

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

Nós também precisamos adicionar a propriedade `parent` à classe do `Child`.

Agora que a entidade `Child` está gerenciando o estado do link, informaremos à coleção para não atualizar o link. Utilizamos o atributo `inverse` para isto:

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
```

```
</set
>
```

O seguinte código seria usado para adicionar um novo `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 SQL `INSERT` seria editado.

Para assegurar tudo isto, podemos criar um método de `addChild()` do `Parent`.

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

Agora, o código que adiciona um `Child` se parece com este:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

24.3. Ciclo de vida em Cascata

A chamada explícita para `save()` ainda é incômoda. Iremos nos referir a ela utilizando cascatas.

```
<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();
```

Da mesma forma, não precisamos repetir este comando com os filhos ao salvar ou deletar um `Parent`. O comando seguinte irá remover o `p` e todos os seus filhos do banco de dados.

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

No entanto, 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 irá remover `c` do banco de dados. Neste caso, ele somente removerá o link para `p` e causará uma violação de restrição `NOT NULL`). Você precisará `delete()` de forma explícita o `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, no seu caso, um `Child` não pode existir sem seu pai. Então, se removermos um `Child` da coleção, não iremos mais querer que ele seja deletado. Devido a isto, devemos utilizar um `cascade="all-delete-orphan"`.

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set
>
```

Apesar do mapeamento da coleção especificar `inverse="true"`, as cascatas ainda são processadas por repetição dos elementos de coleção. Portanto, se você requiser que um objeto seja salvo, deletado ou atualizado por uma cascata, você deverá adicioná-lo à sua coleção. Chamar `setParent()` não é o bastante.

24.4. Cascatas e `unsaved-value`

Suppose we loaded up a `Parent` in one `Session`, made some changes in a UI action and wanted to persist these changes in a new session by calling `update()`. The `Parent` will contain a collection of children and, since the cascading update is enabled, Hibernate needs to know which children are newly instantiated and which represent existing rows in the database. We will also assume that both `Parent` and `Child` have generated identifier properties of type `Long`. Hibernate will use the identifier and version/timestamp property value to determine which of the children are new. (See *Seção 11.7, "Detecção automática de estado"*.) *In Hibernate3, it is no longer necessary to specify an `unsaved-value` explicitly.*

O seguinte código atualizará o `parent` e o `child` e inserirá um `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, isto cabe bem no caso de um identificador gerado, mas e os identificadores atribuídos e os identificadores de composição? Isto é mais difícil, pois uma vez que o Hibernate não pode utilizar a propriedade do identificador para distinguir entre um objeto instanciado recentemente, com um identificador atribuído pelo usuário, e um objeto carregado em uma sessão anterior. Neste caso, o Hibernate usará tanto um carimbo de data e hora (timestamp) ou uma propriedade de versão, ou irá na verdade consultar um cache de segundo nível, ou no pior dos casos, o banco de dados, para ver se a linha existe.

24.5. Conclusão

Há muito o que digerir aqui e pode parecer confuso na primeira vez. No entanto, na prática, funciona muito bem. A maioria dos aplicativos do Hibernate utiliza o modelo pai/filho em muitos lugares.

Nós mencionamos uma alternativa neste primeiro parágrafo. Nenhum dos casos acima existem no caso de mapeamentos `<composite-element>`, que possuem exatamente a semântica do relacionamento pai/filho. Infelizmente, existem duas grandes limitações para elementos compostos: elementos compostos podem não possuir coleções e assim sendo podem não ser filhos de nenhuma outra entidade a não ser do pai único.

Exemplo: Aplicativo Weblog

25.1. Classes Persistentes

As classes persistentes representam um weblog, e um item postado em um weblog. Eles não devem ser modelados como um relacionamento padrão pai/filho, mas usaremos uma bolsa ordenada ao invés de um conjunto:

```
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;
    }
}
```

25.2. Mapeamentos Hibernate

Os mapeamentos XML devem agora ser um tanto diretos. Por exemplo:

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

25.3. Código Hibernate

A seguinte classe demonstra algumas atividades que podemos realizar 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"
            );
```

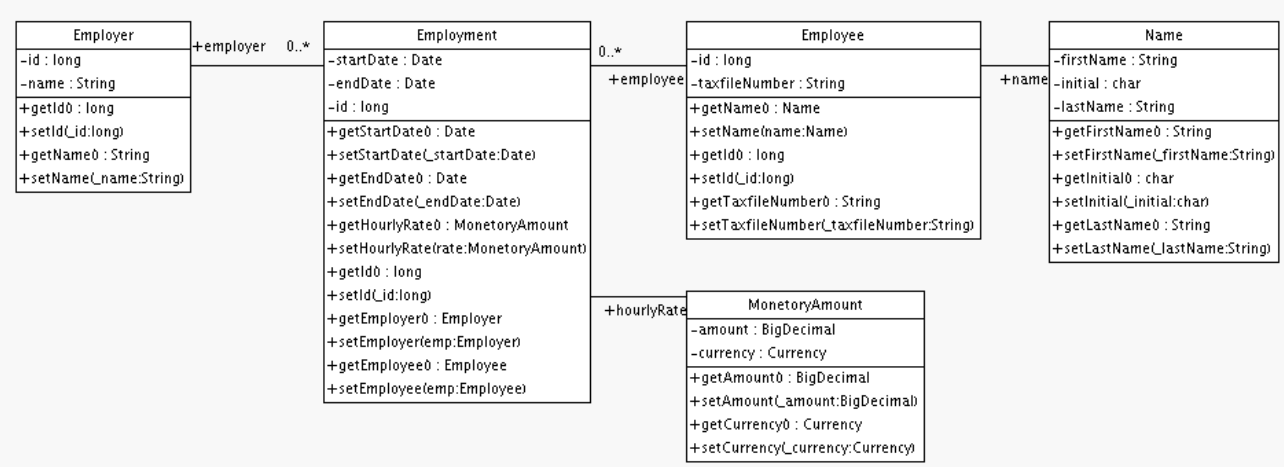
```
            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

Exemplo: Vários Mapeamentos

Este capítulo mostra algums mapeamentos de associações mais complexos.

26.1. Empregador/Empregado

O modelo a seguir, do relacionamento entre `Employer` e `Employee` utiliza uma entidade de classe atual (`Employment`) para representar a associação. Isto é feito porque pode-se 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, segue o documento 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
>
```

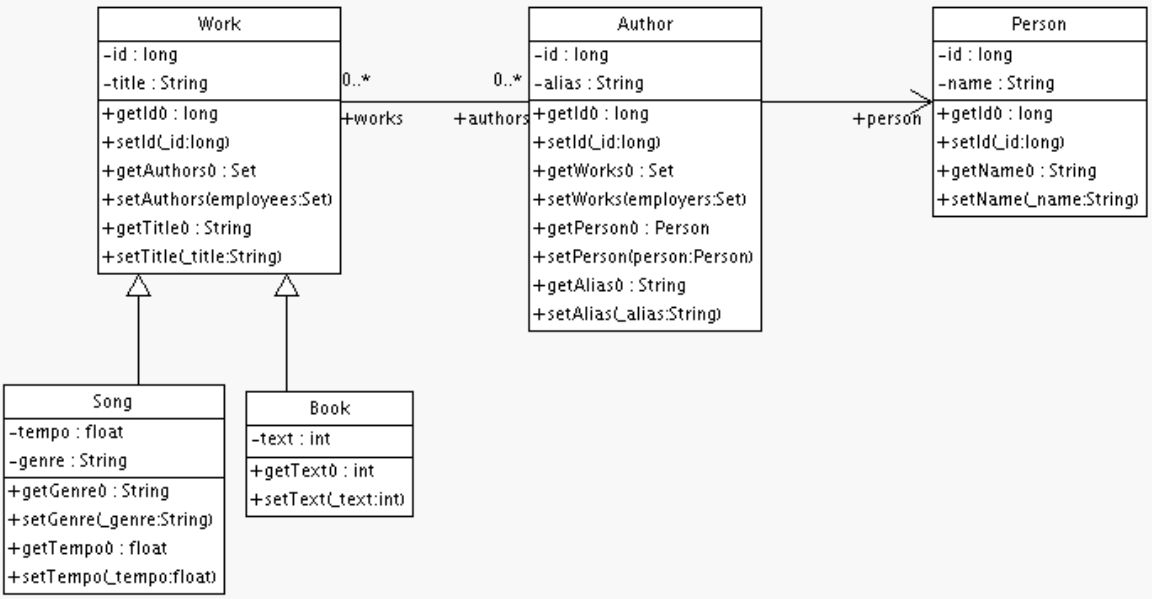
E abaixo, segue o esquema da tabela gerado pelo `SchemaExport`.

```
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
```

26.2. Autor/Trabalho

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` estendendo `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
>
```

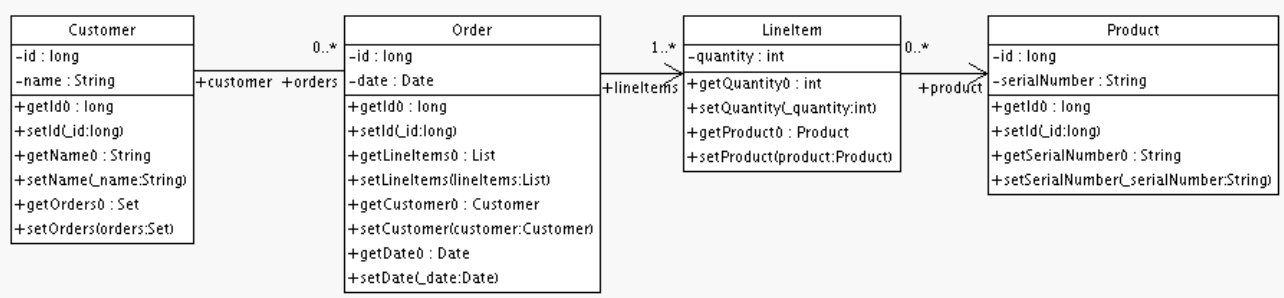
Existem quatro tabelas neste mapeamento: `works`, `authors` e `persons` matém os dados de trabalho, autor e pessoa, respectivamente. O `author_work` é uma tabela de associação que liga autores à trabalhos. Abaixo, segue o esquema das tabelas, gerados pelo `SchemaExport`:

```
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
```

26.3. Cliente/Ordem/Produto

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`? Neste exemplo, o LineItem é mapeado 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 documento de mapeamento será parecido com:

```
<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 ordens a produtos.

```
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
```

26.4. 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 diretório `test` da distribuição do Hibernate.

26.4.1. Associação um-para-um "Typed"

```
<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
>
```

26.4.2. 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
```

```
>
```

26.4.3. Muitos-para-muitos com função 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>
```

26.4.4. Conteúdo baseado em discriminação

```
<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
>
```

26.4.5. 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
>
```

Melhores práticas

Escreva classes compactas e mapeie-as usando `<component>`:
: Use uma classe `Endereço` 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 inúmeras razões para utilizá-las. Nós recomendamos que os identificadores sejam 'sintéticos', quer dizer, gerados sem significado para negócios.

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 faz bastante sentido, especialmente em ambiente de equipe.

Carregue os mapeamentos como recursos:
: Implemente os mapeamentos junto às classes que eles mapeiam.

Considere a possibilidade de externar as strings de consultas:
: Esta é uma boa prática se suas consultas chamam funções SQL que não sejam ANSI. Externar as strings de consultas para mapear arquivos irá tornar a aplicação mais portável.

Use variáveis de vínculo.
: Assim como em JDBC, sempre substitua valores não constantes por "?". Nunca use a manipulação de strings para concatenar valores não constantes em uma consulta. Até melhor, considere a possibilidade de usar parâmetros nomeados nas consultas.

Não gerencie suas conexões JDBC:
: O Hibernate permite que a aplicação gerencie conexões JDBC, mas 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 a possibilidade de usar tipos customizados:
: Suponha que você tenha um tipo Java, de alguma biblioteca, que precisa ser persistido mas não provê de 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 afunilamentos:
: 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 afunilamento. 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, embrulhar a sua operação JDBC como um objeto `org.hibernate.jdbc.Work` 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 esvaziamento da `Session`:
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 a liberação desnecessária desabilitando a liberação automática 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 a arquitetura do bean de sessão/servlet, você pode passar os objetos persistentes carregados no bean de sessão para e a partir da camada servlet/JSP. Use uma nova sessão para manipular cada solicitação. Use a `Session.merge()` ou a `Session.saveOrUpdate()` para sincronizar objetos com o banco de dados.

Em uma arquitetura de duas camadas, considere o uso de contextos de longa persistência:
As Transações do Banco de Dados precisam ser as mais curtas possíveis para uma melhor escalabilidade. No entanto, é geralmente necessário implementar *transações de aplicações* de longa duração, uma única unidade de trabalho a partir do ponto de vista de um usuário. Uma transação de aplicação pode transpor diversos ciclos de solicitação/resposta de cliente. É comum usar objetos desanexados para implementar as transações de aplicação. Uma outra alternativa, extremamente apropriada em uma arquitetura de duas camadas, é manter um único contato de persistência aberto (sessão) para todo o tempo de vida da transação de aplicação e simplesmente disconectá-lo do JDBC ao final de cada solicitação e reconectá-lo no início de uma solicitação subsequente. Nunca compartilhe uma sessão única com mais de uma transação de aplicação, ou você irá trabalhar com dados antigos.

Não trate as exceções como recuperáveis:
Isto é mais uma prática necessária do que uma "melhor" prática. Quando uma exceção ocorre, retorne à `Transaction` e feche a `Sessão`. Se não fizer isto, o Hibernate não poderá garantir que o estado em memória representará de forma precisa o estado persistente. Como este é um caso especial, não utilize a `Session.load()` para determinar se uma instância com dado identificador existe em um banco de dados, use `Session.get()` ou então uma consulta.

Prefira a busca lazy para associações:
Use a busca antecipada de forma moderada. Use as coleções proxy e lazy para a maioria das associações para classes que possam não ser completamente mantidas em cache de segundo nível. Para associações de classes em cache, onde existe uma enorme probabilidade de coincidir caches, desabilite explicitamente a busca antecipada usando `lazy="false"`. Quando uma busca de união é apropriada para um caso específico, use a consulta com `left join fetch`.

Use o modelo *sessão aberta na visualização*, ou uma *fase de construção* para evitar problemas com dados não encontrados.

O Hibernate libera o desenvolvedor de escrever *Objetos de Transferência de Dados* (DTO). Em uma arquitetura tradicional EJB, os DTOs servem dois propósitos: primeiro, eles se deparam com o problema de que os beans de entidade não são serializáveis, depois, eles implicitamente definem uma fase de construção onde todos os dados a serem utilizados pelo view são buscados e conduzidos aos DTOs antes mesmo de retornar o controle à camada de apresentação. O Hibernate elimina o primeiro propósito. No entanto, você ainda precisará de uma fase de construção (pense em seus métodos de negócios como tendo um contrato estrito com a camada de apresentação sobre o quais dados estão disponíveis nos objetos desanexados) a não ser que você esteja preparado para manter o contexto de persistência (sessão) aberto no processo de renderização da visualização. Isto não é uma limitação do Hibernate. É uma solicitação fundamental para acesso a dados transacionais seguros.

Considere abstrair sua lógica comercial do Hibernate:

Oculte (Hibernate) o código de acesso a dados atrás de uma interface. Combine os modelos *DAO* e *Sessão Local de Thread*. Você pode também persistir algumas classes pelo JDBC handcoded, associado ao Hibernate via um `UserType`. Este é um conselho para aplicações "grandes o suficiente", não é apropriado para uma aplicação com cinco tabelas.

Não use mapeamentos de associação exóticos:

Casos de testes práticos para associações muitos-para-muitos reais são raros. A maioria do tempo você precisa de informação adicional armazenada na " tabela de link". Neste caso, é muito melhor usar associações dois um-para-muitos para uma classe de link intermediário. Na verdade, acreditamos que a maioria das associações é um-para-muitos e muitos-para-um, você deve tomar cuidado ao utilizar qualquer outro tipo de associação e perguntar a você mesmo se é realmente necessário.

Prefira associações bidirecionais:

As associações unidirecionais são mais difíceis para pesquisar. Em aplicações grandes, quase todas as associações devem navegar nas duas direções em consultas.

Considerações da Portabilidade do Banco de Dados

28.1. Fundamentos da Portabilidade

Um dos pontos de venda do Hibernate (e realmente Mapeamento do Objeto/Relacional como um conjunto) é a noção da portabilidade do banco de dados. Isto pode significar um usuário de TI interno migrando a partir de um fornecedor de banco de dados a outro, ou isto pode significar que um framework ou aplicativo implementável consumindo o Hibernate para produtos de banco de dados múltiplos de destinação simultaneamente pelos usuários. Independente do cenário exato, a idéia básica é que você queira que o Hibernate o ajude a rodar em referência a qualquer número de banco de dados sem as alterações a seu código e preferencialmente sem quaisquer alterações ao metadados de mapeamento.

28.2. Dialeto

A primeira linha de portabilidade para o Hibernate é o dialeto, que trata-se de uma especialização de um contrato `org.hibernate.dialect.Dialect`. Um dialeto encapsula todas as diferenças em como o Hibernate deve comunicar-se com um banco de dados particular para completar algumas tarefas como obter um valor de seqüência ou estruturar uma consulta SELECT. O Hibernate vincula uma variedade de dialetos para muitos dos bancos de dados mais populares. Se você achar que seu banco de dados particular não está seguindo os mesmos, não será difícil escrever o seu próprio.

28.3. Resolução do Dialeto

Originalmente, o Hibernate sempre solicita que os usuários especifiquem qual dialeto a ser usado. No caso dos usuários buscarem banco de dados múltiplos de destinação simultaneamente com as próprias construções que eram problemáticas. Normalmente, isto solicita que seus próprios usuários configurem o dialeto do Hibernate ou definam o próprio método de determinação do valor.

Inicializando com a versão 3.2, o Hibernate introduziu a noção de detecção automática do dialeto para uso baseado no `java.sql.DatabaseMetaData` obtido a partir de um `java.sql.Connection` para aquele banco de dados. Era muito melhor, esperar que esta resolução limitada aos bancos de dados Hibernate soubesse com antecedência e que em ocasião alguma era configurável ou substituível.

Starting with version 3.3, Hibernate has a fare more powerful way to automatically determine which dialect to should be used by relying on a series of delegates which implement the `org.hibernate.dialect.resolver.DialectResolver` which defines only a single method:

```
public Dialect resolveDialect(DatabaseMetaData metaData) throws JDBCConnectionException
```

The basic contract here is that if the resolver 'understands' the given database metadata then it returns the corresponding Dialect; if not it returns null and the process continues to the next resolver. The signature also identifies `org.hibernate.exception.JDBCConnectionException` as possibly being thrown. A JDBCConnectionException here is interpreted to imply a "non transient" (aka non-recoverable) connection problem and is used to indicate an immediate stop to resolution attempts. All other exceptions result in a warning and continuing on to the next resolver.

A melhor parte destes solucionadores é que os usuários também podem registrar os seus próprios solucionadores personalizados dos quais serão processados antes dos Hibernates internos. Isto poderá ser útil em um número diferente de situações: permite uma integração fácil de auto-detecção de dialetos além daqueles lançados com o próprio Hibernate. Além disto, permite que você especifique o uso de um dialeto personalizado quando um banco de dados particular é reconhecido, etc. Para registrar um ou mais solucionadores, apenas especifique-os (separados por vírgula, tabs ou espaços) usando o conjunto de configuração 'hibernate.dialect_resolvers' (consulte a constante `DIALECT_RESOLVERS` no `org.hibernate.cfg.Environment`).

28.4. Geração do identificador

When considering portability between databases, another important decision is selecting the identifier generation stratagy you want to use. Originally Hibernate provided the *native* generator for this purpose, which was intended to select between a *sequence*, *identity*, or *table* strategy depending on the capability of the underlying database. However, an insidious implication of this approach comes about when targtetting some databases which support *identity* generation and some which do not. *identity* generation relies on the SQL definition of an IDENTITY (or auto-increment) column to manage the identifier value; it is what is known as a post-insert generation strategy becauase the insert must actually happen before we can know the identifier value. Because Hibernate relies on this identifier value to uniquely reference entities within a persistence context it must then issue the insert immediately when the users requests the entitiy be associated with the session (like via save() e.g.) regardless of current transactional semantics.

Nota

Hibernate was changed slightly once the implication of this was better understood so that the insert is delayed in cases where that is feasible.

The underlying issue is that the actual semanctics of the application itself changes in these cases.

Starting with version 3.2.3, Hibernate comes with a set of *enhanced* [http://in.relation.to/2082.lace] identifier generators targetting portability in a much different way.

Nota

There are specifically 2 bundled *enhanced*generators:

- `org.hibernate.id.enhanced.SequenceStyleGenerator`
- `org.hibernate.id.enhanced.TableGenerator`

The idea behind these generators is to port the actual semantics of the identifer value generation to the different databases. For example, the `org.hibernate.id.enhanced.SequenceStyleGenerator` mimics the behavior of a sequence on databases which do not support sequences by using a table.

28.5. Funções do banco de dados

Atenção

Esta é uma área do Hibernate com necessidade de melhoramentos. Este manuseio de função funciona atualmente muito bem com o HQL, quando falamos das preocupações de portabilidade. No entanto, é bastante precária em outros aspectos.

As funções SQL podem ser referenciadas em diversas maneiras pelos usuários. No entanto, nem todos os bancos de dados suportam o mesmo conjunto de função. O Hibernate fornece um significado de mapeamento do nome da função *lógica* para uma delegação que sabe como manusear aquela função em particular, mesmo quando usando uma chamada de função física totalmente diferente.

Importante

Technically this function registration is handled through the `org.hibernate.dialect.function.SQLFunctionRegistry` class which is intended to allow users to provide custom function definitions without having to provide a custom dialect. This specific behavior is not fully completed as of yet.

It is sort of implemented such that users can programatically register functions with the `org.hibernate.cfg.Configuration` and those functions will be recognized for HQL.

28.6. Tipos de mapeamentos

A seção está esquematizada para finalização numa data posterior...

Referências

[PoEAA] *Padröes da Arquitetura do Aplicativo Enterprise* . 0-321-12742-0. por Martin Fowler. Copyright © 2003 Pearson Education, Inc.. Addison-Wesley Publishing Company.

[JPwH] *Persitência Java com Hibernate*. Segunda Edição do Hibernate em Ação. 1-932394-88-5. *http://www.manning.com/bauer2* . por Christian Bauer e Gavin King. Copyright © 2007 Manning Publications Co.. Manning Publications Co..