Hibernate开发者指南

Hibernate团队

JBoss视觉设计团队

4 1 11最后

2013-03-18

表的内容

表的列表

列表的例子

前言

表的内容

使用面向对象的软件和关系数据库可以繁琐和费时。 开发成本明显上涨,因一个范式之间不匹配数据的代表 对象与关系数据库。 Hibernate是一个对象/关系映射为Java环境的解决方案。 术语对象/关系映射指的是技术的映射数据从一个对象模型表示 到关系数据模型表示(和签证,反之亦然)。 看到 http://en.wikipedia.org/wiki/Object-relational_mapping 对于一个好的高层讨论。

> **注意**
>
> 虽然有一个强大的背景在SQL中不需要使用Hibernate,有一个基本的了解 这个概念可以极大地帮助您了解更全面和快速。冬眠 可能单一 最好的背景是一个理解数据建模原则。 你可能想考虑这些资源 作为一个良好的起点:
>
> - http://www.agiledata.org/essays/dataModeling101.html
> - http://en.wikipedia.org/wiki/Data_modeling

Hibernate不仅负责从Java类映射到数据库表(和从Java数据类型 SQL数据类型),但也提供了数据查询和检索设施。 它可以显著降低 开发时间否则花手工数据处理在SQL和JDBC。 Hibernate的设计目标是 减轻开发人员从95%的通用数据持续相关编程任务通过消除需要 手册,手工数据处理使用SQL和JDBC。 然而,与许多其他持久性的解决方案, Hibernate不隐藏的力量,从你和确保SQL投资关系的技术 和知识是作为有效的一如既往。

Hibernate可能不是最佳的解决方案为以数据为中心的应用程序只使用存储过程来 实现业务逻辑在数据库中,它是最有用的和面向对象的领域模型和业务 逻辑在基于java的中间层。 然而,Hibernate当然可以帮助你去除或封装 特定于供应商的SQL代码和将帮助常见任务的结果集转换从一个表格 表示对图形对象。

1。 参与

- 使用Hibernate和报告任何错误或问题你发现。 看到 http://hibernate.org/issuetracker.html 详情。
- 试着动手修复一些错误或实施改进。 再次,看到 http://hibernate.org/issuetracker.html 。
- 参与社区使用邮件列表、论坛、IRC或其他方式标价 http://hibernate.org/community.html 。
- 帮助改善或翻译这个文件。 联系我们在 开发人员邮件列表,如果你有兴趣。
- 传播这个消息。 让你其他的组织知道的好处 Hibernate。

2。 入门指南

新用户可能需要先浏览一下 **Hibernate入门指南** 基本信息以及 教程。 即使是经验丰富的老兵可能想要考虑在阅读部分用于修饰或说明 构建工件有无变化。

第1章。 数据库访问

表的内容

1.1。连接

Hibernate数据库连接到应用程序的代表。 它可以连接通过多种机制, 包括:

- 独立内置连接池
- javax.sql.DataSource
- 连接池,包括支持两种不同的第三方开源JDBC连接池:
  - C3P0
  - proxool
- 应用程序提供的xmleventallocator JDBC连接。 这不是一个推荐的方法和存在的遗留问题

> **注意**
>
> 内置的连接池是不能用于生产环境。

Hibernate获得JDBC连接所需要的尽管 org.hibernate.service.jdbc.connections.spi.ConnectionProvider 接口 这是一个服务合同。 应用程序也可以提供他们自己的 org.hibernate.service.jdbc.connections.spi.ConnectionProvider 实现 定义一个自定义的方法来提供连接Hibernate(从一个不同的连接池 实现,例如)。

1.1.1。配置

您可以配置数据库连接使用一个属性文件,XML部署描述符或 以编程方式。

例1.1。 hibernate属性 对于一个c3p0连接池

```
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.PostgreSQL82Dialect
```

例1.2。 hibernate cfg xml 对于连接到捆绑HSQL数据库

```
<?xml version='1.0' encoding='utf-8'?>

<hibernate-configuration
      xmlns="http://www.hibernate.org/xsd/hibernate-configuration"
      xsi:schemaLocation="http://www.hibernate.org/xsd/hibernate-configuration hibernate-configuration-
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <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.internal.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>
```

1 1 1 1。 带来了有步骤配置

对象的一个实例 org hibernate cfg配置 代表整个组 映射一个应用程序的Java类型到一个SQL数据库。 这个 org hibernate cfg配置 构建一个不变的 org hibernate sessionfactory 和编译映射从不同的XML映射 文件。 你可以指定映射文件直接或Hibernate可以找到给你。

例1.3。 直接指定映射文件

你可以获得一个 org hibernate cfg配置 实例通过实例化它 直接和指定的XML映射文件。 如果映射文件在类路径中,使用方法 addResource() 。

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

例1.4。 让Hibernate映射文件为你找到

这个 选择用addClass() 方法指导Hibernate来搜索类路径的映射 文件,消除硬编码的文件名称。 在接下来的例子中,它将搜索 **org/hibernate/auction/Item.hbm.xml** 和 **org/hibernate/auction/Bid.hbm.xml** 。

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

例1.5。 指定配置属性

```
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");
```

其他方法来配置Hibernate编程方式

- 通过一个实例, java util属性 到 配置找() 。
- 设置系统属性使用 **JAVA - d 财产 = 价值**

1.1.2。 获得JDBC连接

在你配置 最重要的Hibernate JDBC属性 ,你可以使用方法 openSession 类 org hibernate sessionfactory 打开 会话。 会议将根据需要获得JDBC连接基于提供的配置。

例1.6。 指定配置属性

```
Session session = sessions.openSession();
```

最重要的Hibernate JDBC属性

- hibernate连接驱动程序类
- hibernate连接url
- hibernate连接用户名
- hibernate连接密码
- hibernate连接池大小

所有可用的Hibernate设置作为常量定义和讨论的 org.hibernate.cfg.AvailableSettings 接口。 看到自己的源代码 JavaDoc 详情。

1.2。 连接池

Hibernate的内部连接池算法是基本的,是供开发和测试 目的。 使用第三方池为最佳性能和稳定性。 使用一个第三方的池,更换 hibernate连接。 池大小属性 与设置特定于您的连接池 的选择。 这将禁用Hibernate的内部连接池。

1.2.1。 c3p0连接池

C3P0是一个开源的JDBC连接池分布随着Hibernate在 **lib** **/** 目录。 Hibernate使用它 org.hibernate.service.jdbc.connections.internal.C3P0ConnectionProvider 对于 如果您设置的连接池 hibernate c3p0。 * 属性。 属性。

重要配置属性c3p0连接池

- hibernate c3p0最小尺寸
- hibernate c3p0马克斯大小
- hibernate c3p0超时
- hibernate c3p0马克斯语句

1.2.2。 Proxool连接池

Proxool是另一个开放源码的JDBC连接池分布随着Hibernate在 **lib** **/** 目录。 Hibernate使用它 org.hibernate.service.jdbc.connections.internal.ProxoolConnectionProvider 如果你用于连接池设置 hibernate.proxool。* 属性。 不像c3p0,proxool需要一些额外的配置 参数,如文档所描述的Proxool可用 http://proxool.sourceforge.net/configure.html 。

表1.1。 重要的配置属性Proxool连接池

财产	描述
hibernate.proxool.xml	配置Proxool提供者使用一个XML文件(。 xml是附加自动)
hibernate.proxool.properties	配置Proxool提供者使用一个属性文件(。 属性是附加 自动)
hibernate.proxool.existing_pool	是否配置Proxool提供者从现有的池
hibernate.proxool.pool_alias	Proxool池别名使用。 要求。

1.2.3。 从应用程序服务器获取连接,使用JNDI

使用Hibernate在一个应用程序服务器,配置Hibernate来获取连接从一个应用程序 服务器 javax.sql.DataSource 注册在 JNDI,通过设置至少一个以下 属性:

重要的Hibernate属性JNDI数据源

- hibernate连接。 数据源(必需)
- hibernate jndi url
- hibernate jndi类
- hibernate连接用户名
- hibernate连接密码

JDBC连接得到一个JNDI数据源自动参与容器管理的事务 应用程序服务器。

1.2.4。 其他连接特定的配置

你可以通过任意的连接属性我们可以通过按下 hibernate连接 到 连接属性的名字。 例如,指定字符集连接属性 hibernate连接字符集 。

您可以定义您自己的插件策略获得JDBC连接通过实现接口 org.hibernate.service.jdbc.connections.spi.ConnectionProvider 并指定您的自定义 实现与 hibernate连接provider类 财产。

1 2 5。 可选配置属性

除了提到的属性在前面的章节中,包括许多其他可选。冬眠 属性。 看到 吗？？？ 为一个更完整的列表。

1.3。方言

虽然SQL相对标准化,每个数据库供应商使用一个子集的支持语法。 这被称为 作为一个 方言 。 Hibernate处理跨这些方言变体通过它 org hibernate方言方言 类和子类对于每个供应商的各种方言。

表1.2。 支持数据库方言

数据库	方言
db2	org.hibernate.dialect.DB2Dialect
DB2 / 400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
火鸟	org.hibernate.dialect.FirebirdDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Ingres	org.hibernate.dialect.IngresDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQL5InnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle 8i	org.hibernate.dialect.Oracle8iDialect
Oracle 9 i	org.hibernate.dialect.Oracle9iDialect
Oracle 10 g	org.hibernate.dialect.Oracle10gDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
PostgreSQL 8.1	org.hibernate.dialect.PostgreSQL81Dialect
PostgreSQL 8.2及以后	org.hibernate.dialect.PostgreSQL82Dialect
进步	org.hibernate.dialect.ProgressDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Sybase ASE 15.5	org.hibernate.dialect.SybaseASE15Dialect
Sybase ASE 15.7	org.hibernate.dialect.SybaseASE157Dialect
Sybase任何地方	org.hibernate.dialect.SybaseAnywhereDialect

1.3.1。 指定要使用的方言

开发人员可以手动指定方言使用通过设置 hibernate方言 配置属性名称的一个特定的 org hibernate方言方言 类来使用。

1.3.2。 方言决议

假设一个 org.hibernate.service.jdbc.connections.spi.ConnectionProvider 一直 设置时,Hibernate会自动确定方言使用基于 java sql该方法 报告 java sql连接 所得, org.hibernate.service.jdbc.connections.spi.ConnectionProvider 。

这个功能是由一系列的 org.hibernate.service.jdbc.dialect.spi.DialectResolver 实例注册 在内部与Hibernate。 Hibernate附带一组标准的认识。 如果您的应用程序 需要额外的方言分辨率能力,它只会注册一个自定义的实现 的 org.hibernate.service.jdbc.dialect.spi.DialectResolver 如下:

注册 org.hibernate.service.jdbc.dialect.spi.DialectResolver 是 前缀 到一个内部解析器的列表,所以他们优先考虑 之前已经注册解析器包括标准的一个。

1.4。 自动模式生成与SchemaExport

是一个Hibernate SchemaExport工具生成DDL从你的映射文件。 生成的模式包括 引用完整性约束、主键和外键,对于实体和收集表。 它还创建了 表和序列映射标识符生成器。

> **注意**
>
> 您必须指定一个SQL方言通过 hibernate方言 房地产当使用这个工具，因为DDL高度特定于供应商的。 看到 1.3节，“方言” 对于信息。

Hibernate可以生成你的模式之前,你必须定制你的映射文件。

1.4.1。 定制映射文件

Hibernate提供了几个元素和属性来定制你的映射文件。 它们列在 表格1.3,“元素和属性提供了自定义映射文件” ,和一个定制的逻辑顺序,提出了 程序1.1,“定制模式” 。

表1.3。 元素和属性提供了定制映射文件

名称	类型的值	描述
长度	号码	柱长度
精密	号码	小数精度的列
规模	号码	十进制的列
过的非null的	真正的 或 假	这一列是否被允许保持null值
独特的	真正的 或 假	列中的值是否必须是唯一的
指数	字符串	多列索引的名称
独特的关键	字符串	多列的名称唯一约束
外键	字符串	这个名字的外键约束生成一个协会。 这适用于 <一对一>,<多对一的>,<键>,和<多对多>映射 元素。 逆=＂true＂双方都由SchemaExport跳过。
sql类型	字符串	覆盖默认的列类型。 这适用于<列>元素只有。
默认	字符串	默认的列的值
检查	字符串	一个SQL检查约束可以在一个列或就餐

程序1.1。 定制模式

1. **设置长度、精度和尺度映射元素。**

   许多Hibernate映射元素定义可选属性命名 **长度** , **精密** ,和 **规模** 。

   ```
   <property name="zip" length="5"/>
   <property name="balance" precision="12" scale="2"/>
   ```

2. **设置 过的非null的 , 独特的 , 独特的关键 属性。**

   这个 **过的非null的** 和 **独特的** 属性生成表的列上的限制。

   这个独特的关键属性组列在一个单一的、独特的关键约束。 目前,指定的值 的独特的关键属性不命名约束在生成的DDL。 它只组列 映射文件。

   ```
   <many-to-one name="bar" column="barId" not-null="true"/>
   <element column="serialNumber" type="long" not-null="true" unique="true"/>

   <many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
   <property name="employeeId" unique-key="OrgEmployee"/>
   ```

3. **设置 指数 和 外键 属性。**

   这个 **指数** 属性指定一个索引的名称为Hibernate创建使用映射 列或列。 你可以把多个列成相同的索引分配他们相同的索引名称。

   一个外键属性将覆盖任何生成的名字外键约束。

   ```
   <many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
   ```

4. **集孩子 <列> 元素。**

   许多映射元素接受一个或多个子<列>元素。 这是特别有用的 映射类型涉及多个列。

   ```
   <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>
   ```

5. **设置 默认 属性。**

   这个 **默认** 属性代表一个默认值为一个列。 分配相同的值 映射属性在保存之前的一个新实例的映射类。

   ```
   <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>
   ```

6. **设置 sql类型 attribure。**

   使用 **sql类型** 属性来覆盖默认的映射到SQL的一个Hibernate类型 数据类型。

   ```
   <property name="balance" type="float">
   ```

```
    <column name="balance" sql-type="decimal(13,3)"/>
</property>
```

7. **设置 检查 属性。**

   使用 **检查** 属性来指定一个 检查 约束。

```
<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>
```

8. **<评论>元素添加到你的模式。**

   使用<评论>元素指定注释生成的模式。

```
<class name="Customer" table="CurCust">
  <comment>Current customers only</comment>
  ...
</class>
```

1.4.2。 运行SchemaExport工具

SchemaExport工具写一个DDL脚本到标准输出,执行DDL语句,或两者兼而有之。

例1.7。 SchemaExport语法

```
java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaExport options mapping_files
```

表1.4。 SchemaExport选项

选项	描述
——安静	不输出到标准输出的脚本吗
——降	只有把表
——创建	只有创建表
——文本	不出口到数据库吗
——输出= **我的模式ddl**	输出到一个文件的ddl脚本
——命名= **eg.MyNamingStrategy**	选择一个NamingStrategy
——config = **hibernate cfg xml**	读Hibernate配置从一个XML文件
——属性= **hibernate属性**	从文件读取数据库属性
——格式	格式生成的SQL脚本的到位
——分隔符= **;**	设置一个回车分隔符为脚本

例1.8。 SchemaExport嵌入到应用程序中

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

第二章。 事务和并发控制

表的内容

2.1。 定义事务

重要的是要明白这个词有很多不同但是相关事务的含义在问候 持久性和对象/关系映射。 在大多数用例这些定义对齐,但这不是 总是如此。

- 可能引用物理事务与数据库。
- 可能指的是逻辑概念的一个事务有关的一个持久化上下文。
- 可能引用应用程序工作单元的概念所定义的原型模式。

> **注意**
>
> 本文档主要治疗身体和逻辑概念的事务作为一个在相同的。

2.2。 物理事务

Hibernate使用JDBC API,用于持久性。 Java世界中有两个定义良好的机制 处理事务在JDBC:JDBC本身和JTA。 Hibernate既支持机制 结合事务和允许应用程序来管理物理事务。

第一个概念是理解Hibernate事务支持 org.hibernate.engine.transaction.spi.TransactionFactory 接口, 服务2主要功能:

- 它允许Hibernate来理解环境的事务语义。 是我们操作 在一个JTA环境? 是一个物理事务已经目前活跃? 等。
- 它作为一个工厂 org hibernate事务 实例, 是用来允许应用程序管理和检查事务的状态。 org hibernate事务 是Hibernate的概念逻辑 事务。 JPA有类似的概念 javax.persistence.EntityTransaction 接口。

> **注意**
>
> javax.persistence.EntityTransaction 只能在使用吗 资源本地事务。 Hibernate允许访问org hibernate事务 不管环境。

org.hibernate.engine.transaction.spi.TransactionFactory 是一个标准的 Hibernate服务。 看到 节7 5 16。" org.hibernate.engine.transaction.spi.TransactionFactory " 详情。

2.2.1。 物理事务- JDBC

基于JDBC事务管理利用JDBC定义方法 java sql连接commit() 和 java sql连接rollback() (JDBC没有定义一个显式的 开始一个事务)的方法。 在冬眠,这方法是为代表的 org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory 类。

2.2.2。 物理事务——JTA

基于jta事务的方法,这种方法利用了 javax.transaction.UserTransaction 接口获得 org.hibernate.service.jta.platform.spi.JtaPlatform API。 这种方法 代表的 org.hibernate.engine.transaction.internal.jta.JtaTransactionFactory 类。

看到 节7 5 9。" org.hibernate.service.jta.platform.spi.JtaPlatform " 信息集成与底层JTA 系统。

2.2.3。 物理事务——CMT

另一种方法,这种方法基于JTA事务利用JTA javax.transaction.TransactionManager 接口获得 org.hibernate.service.jta.platform.spi.JtaPlatform API。 这种方法 代表的 org.hibernate.engine.transaction.internal.jta.CMTTransactionFactory 类。 在 一个实际的JEE CMT环境,访问 javax.transaction.UserTransaction 是受限制的。

> **注意**
>
> 这个词可能是误导这里CMT。 重要的一点是,简单的物理JTA 事务是要由其他东西Hibernate事务API。

看到 节7 5 9。" org.hibernate.service.jta.platform.spi.JtaPlatform " 信息集成与底层JTA 系统。

2 2 4。 物理事务——自定义

它还可以塞在一个自定义事务的方法实现 org.hibernate.engine.transaction.spi.TransactionFactory 合同。 默认的服务引发剂有内置支持理解自定义事务的方法 通过 hibernate事务工厂类 可以的名字不是:

- 的实例 org.hibernate.engine.transaction.spi.TransactionFactory 使用。
- 类的名称实施 org.hibernate.engine.transaction.spi.TransactionFactory 使用。 我们的期望是实现类有一个无参数的构造函数。

2.2.5。 物理事务——遗留

在开发4.0,大多数这些类命名这里被转移到新的包。 帮助 方便升级,Hibernate将认识到遗留的名字在这里很短的一段时间。

- org.hibernate.transaction.JDBCTransactionFactory 映射到 org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory
- org.hibernate.transaction.JTATransactionFactory 映射到 org.hibernate.engine.transaction.internal.jta.JtaTransactionFactory
- org.hibernate.transaction.CMTTransactionFactory 映射到 org.hibernate.engine.transaction.internal.jta.CMTTransactionFactory

2.3。 Hibernate事务使用

Hibernate使用JDBC连接和JTA资源直接,没有添加任何额外的锁定行为。 它是重要为你熟悉JDBC,ANSI SQL,事务隔离的细节 你的数据库管理系统。

Hibernate不锁对象在内存中。 行为定义为您的数据库的隔离级别 交易不改变当你使用Hibernate。 Hibernate org hibernate会话 作为一个以缓存提供 可重复读取由标识符查找和查询,导致加载实体。

> **重要**
>
> 减少数据库中的锁争用,物理数据库事务需要那么短 可能的。 长数据库事务防止应用程序从扩展到一个高度并发负载。 不持有一个数据库事务公开在最终用户级别的工作,但是打开之后最终用户级别 工作完成了。 这是概念称为 事务写入后 。

2.4。 事务性模式(和反模式)

2 4 1。 会话/操作反模式

这是一种反模式的开启和关闭 会话 对于每个数据库调用 在一个线程。 它也是一种反模式从数据库事务。 集团数据库 调用一个计划序列。 同样,不自动提交的每个SQL语句之后在你的 应用程序。 Hibernate禁用,或预计应用程序服务器禁用,自动提交模式 立即。 数据库事务是从来没有可选的。 所有的通信与数据库必须 被封装在一个事务。 避免自动提交行为阅读数据,因为许多小 交易不太可能表现得比一个清晰明确的工作单元,和更 难以维护和扩展。

> **注意**
>
> 使用自动提交不规避数据库事务。 相反,当在自动提交模式。 JDBC驱动程序只执行每个调用一个隐式事务调用。 这就好像你的应用程序 称为提交后每个JDBC调用。

2.4.2。 使用“按请求会话”模式模式

这是最常见的交易模式。 这个词在这里请求涉及系统的概念 反应一系列请求从客户端/用户。 Web应用程序是一个主要的例子 类型的系统,尽管显然不是唯一的。 开始处理这样的要求, 程序打开一个Hibernate 会话 ,启动一个事务,执行 所有的数据相关的工作,结束了交易和关闭 会话 。 的关键模式之间的——对应关系事务和 会话 。

在这个模式有一个共同的技术定义 当前会话 到 简化需要通过这个 会话 在所有的应用程序 组件可能需要访问它。 Hibernate提供了支持这种技术通过 getCurrentSession 方法 SessionFactory 。 这个概念的一个“当前”会议必须有一个范围,定义了边界的概念 “当前”是有效的。 这是目的 org.hibernate.context.spi.CurrentSessionContext 合同。 有2 可靠的定义范围:

- 首先是一个JTA事务,因为它允许回调钩子来知道它何时结束这 让Hibernate机会关闭 会话 和清理。 这是代表的 org.hibernate.context.internal.JTASessionContext 实施 这个 org.hibernate.context.spi.CurrentSessionContext 合同。 使用这种实现,一个 会话 将打开第一 时间 getCurrentSession 被称为在事务。
- 其次是这个应用程序请求周期本身。 这是最好的代表 org.hibernate.context.internal.ManagedSessionContext 实施这个 org.hibernate.context.spi.CurrentSessionContext 合同。 这里一个外部组件负责管理生命周期和范围的一个“当前” 会话。 在开始这样一个范围, ManagedSessionContext ' s 绑定 通过调用方法 会话 。 最后,它的 解开 方法被调用。

  一些常见的例子这样的“外部组件包括:

  - javax.servlet.Filter 实现
  - AOP拦截器与一个切入点的服务方法

- 代理/拦截容器

> **重要**
>
> 这个 getCurrentSession() 方法有一个缺点在JTA环境。 如果 你使用它,在语句连接释放模式也是默认情况下使用的。 由于限制 JTA规范,Hibernate不能自动清理任何未关的 ScrollableResults 或 迭代器 实例返回的 滚动() 或 迭代() 。 释放底层数据库指针调用 ScrollableResults.close() 或 hibernate关闭(迭代器) 显式地从一个 最后 块。

2.4.3。对话

这个 使用 “按请求会话” 模式 模式不是唯一有效的方式设计的工作单元。 许多业务流程需要一个整体的一系列交互与用户,穿插 数据库访问。 在web和企业应用程序,这是不可接受的数据库事务 跨越一个用户交互。 考虑下面的例子:

程序2.1。 长期运行的对话的一个例子

1. 的第一个屏幕将会打开一个对话框。 被用户看到的数据加载到一个特定的 会话 和数据库事务。 用户可以修改对象。
2. 用户使用一个UI元素保存他们的工作在五分钟的编辑。 修改 是持久的。 用户还希望拥有独家访问数据在编辑 会话。

尽管我们有多个数据库访问这里,从的角度来看,用户,这系列的 步骤代表一个单个的工作单元。 有很多方法可以实现这个在您的应用程序。

第一个天真的实现可能会保留 会话 和数据库事务公开 在用户编辑,使用数据库级的锁来防止其他用户修改相同的 数据和保证隔离和原子性。 这是一种反模式,因为锁竞争是一个 这将防止可伸缩性瓶颈在未来。

几个数据库事务是用于实现对话。 在这种情况下,维护 隔离的业务流程成为应用程序层的部分责任。 一个 谈话通常跨越几个数据库事务。 这些多个数据库访问只能 是原子作为一个整体,如果只有其中一个数据库事务(通常是最后一个)存储 更新的数据。 其他所有只读数据。 一个共同的方法来获得这些数据是通过向导式 对话框生成几个请求/响应周期。 Hibernate包括一些特性使这容易 来实现。

自动版本分离对象延长会话	Hibernate可以执行自动为你乐观并发控制。 它可以 自动检测如果并发修改发生在用户思考时间。 检查这个最后的对话。
	如果你决定使用使用 “按请求会话” 模式模式,所有加载实例 在分离的状态在用户思考时间。 Hibernate允许您重新安装 对象和保存修改。 这个模式称为 每个请求的会话与分离对象。 自动版本用于隔离 并发修改。
	Hibernate 会话 可以断开 底层JDBC连接在数据库事务已经提交, 当一个新的客户端连接请求发生时。 此模式被称为会话每次谈话,让即使回贴不必要的。 自动 版本控制是用来隔离并发修改和 会话 将不允许自动刷新, 只有明确。

每个请求的会话与分离对象 和 会话/谈话 各有优点和缺点。

2.4.4。会话/应用程序

讨论很快. .

2.5。 对象身份

一个应用程序可以同时访问相同的持续状态(数据库行)在两个不同的会话。 然而,一个持久化类的一个实例是从不之间共享两个 会话 实例。 两种不同的观念的存在,进入。身份 在这里扮演:数据库身份和JVM的身份。

例2.1。 数据库身份

```
foo.getId().equals( bar.getId() )
```

例2.2。 JVM身份

```
foo==bar
```

依附于一个特定的对象 会话 ,这两个概念是 等效和JVM身份数据库身份保证了Hibernate。 应用程序可能 同时访问一个业务对象与相同的身份在两个不同的会话,这两个 实例实际上是不同的,根据JVM的身份。 冲突的解决方式使用乐观的 方法和自动版本在冲洗/提交时间。

这种方法把责任在Hibernate和数据库的并发性。 它还提供了 最好的可伸缩性,因为昂贵的锁不需要保证身份在单线程单元 的工作。 应用程序不需要同步任何业务对象,只要它维护 一个线程每反模式。 虽然不推荐,在一个 会话 应用程序可以安全地使用 = = 操作符来比较对象。

然而,一个应用程序使用 = = 算子外的 会话 可能引入的问题. . 如果你把两个分离实例到相同的 集 ,他们可能会 使用相同的数据库身份,这意味着它们代表同一个数据库中的行。 他们将不会 保证有相同的JVM身份如果他们是在一个分离的状态。 覆盖等于 和 Hashcode 方法在持久化类,这样 他们有自己的对象概念平等。 从不使用数据库标识符来实现平等。 相反, 使用一个业务键结合的独特,通常不变的,属性。 数据库标识符 如果一个瞬时对象变化是持久的。 如果瞬态实例,连同分离的情况下, 保存在一个 集 ,改变了散列码打破了合同 集 。 属性可以不稳定的业务键比数据库的主键。 你只 需要保证稳定,只要对象是相同的 集 。 这不是一个 Hibernate问题,但涉及到Java实现的对象身份与平等。

2.6。 常见问题

两 会话/用户会话 和 会话/应用程序 反模式是容易受到以下问题。 一些问题可能也会出现在 推荐的模式,所以确保你理解它的含义作出设计决策:之前

- 一个 会话 不是线程安全的。 并行的工作的事情,喜欢 HTTP请求、会话bean,或者摇摆工人,会导致竞态条件如果一个会话 实例共享。 如果你保持你的冬眠 会话 在你的 javax.servlet.http.HttpSession (这是后来的讨论 章),你应该考虑同步访问你 HttpSession ,否则,用户点击重新加载速度不够快可以使用 同样的 会话 在两个并发运行的线程。
- 一个异常抛出的Hibernate意味着你必须回滚数据库事务 并关闭 会话 立即(更详细的讨论 在之后的章节)。 如果你会话 被绑定到应用程序, 你必须停止应用程序。 回滚数据库事务不把你的业务 对象的状态回到开始时的事务。 这意味着数据库状态和业务对象将不同步。 通常这不是一个 问题,因为异常不可采,你将不得不重新开始之后 无论如何回滚。
- 这个 会话 缓存每个对象在一个持续状态 (看着和检查的变化通过Hibernate)。 如果你保持开放的时间长或简单的负载 太多的数据,它会生长,直到你得到一个OutOfMemoryException无休止。 一个解决方案是 叫 clear() 和 驱逐() 管理会话 缓存,但是你应该考虑另一种方式处理 大量的数据如一个存储过程。 Java是完全不合适的工具做这些 种操作。 一些解决方案中所示 第四章, 批处理 。 保持一个 会话 开放期间的用户会话也意味着更高的 概率的失效数据。

第三章。 持久性上下文

表的内容

两 org hibernate会话 API和 javax.persistence.EntityManager API代表一个上下文来处理 持久数据。 这个概念被称为一个 持久化上下文 。 持久数据有一个 国家有关两一个持久化上下文和底层数据库。

实体状态

- 新 ,或 瞬态 ——实体刚刚被实例化,是 没有相关联的持久化上下文。 它没有持久表示在数据库和没有 的标识符值已经被分配。
- 管理 ,或 持久 ——实体都有一个关联的标识符 并且伴随着一个持久化上下文。
- 分离 ——实体都有一个关联的标识符,但不再是相关联的 一个持久上下文(通常因为持久性上下文被关闭或被驱逐的实例从上下文)
- 删除 ——实体都有一个关联的标识符和具有持久性 上下文,然而它定于从数据库中删除。

在Hibernate本机api,持久性上下文被定义为 org hibernate会话 。 在JPA,持久性上下文被定义为 javax.persistence.EntityManager 。 大部分的 org hibernate会话 和 javax.persistence.EntityManager 方法处理移动实体之间这些 州。

3.1。 使实体持续

一旦你创建了一个新的实体实例(使用标准 新 操作符),这在 新 状态。 你可以让它持久的通过将它要么 org hibernate会话 或 javax.persistence.EntityManager

例3.1。 例子使一个实体持久

```
DomesticCat fritz = new DomesticCat();
fritz.setColor( Color.GINGER );
fritz.setSex( 'M' );
fritz.setName( "Fritz" );
session.save( fritz );
```

```
DomesticCat fritz = new DomesticCat();
fritz.setColor( Color.GINGER );
fritz.setSex( 'M' );
fritz.setName( "Fritz" );
entityManager.persist( fritz );
```

org hibernate会话 还有一个方法命名 持续 这是确切的语义定义在JPA规范 持续 法。 正是这种方法 org hibernate会话 而 Hibernate javax.persistence.EntityManager 实现代表。

如果 domesticCat 实体类型有一个生成的标识符,这个值是相关的 到实例当 保存 或 持续 被称为。 如果 标识符不是自动生成的,应用程序分配(通常是自然)键值必须 之前设置实例 保存 或 持续 被称为。

3.2。 删除实体

实体也可以被删除。

例3.2。 删除一个实体的例子

```
session.delete( fritz );
```

```
entityManager.remove( fritz );
```

重要的是要注意,Hibernate本身可以处理删除分离的状态。 JPA,然而,不允许 它。 言下之意是,实体实例传递到 org hibernate会话 删除 方法可以 在管理或分离的状态,而实体实例传递给 删除 在 javax.persistence.EntityManager 必须在管理的状态。

3.3。 获得一个实体引用没有初始化它的数据

有时被称为延迟加载,能够获得一个参考实体无需 负荷数据是非常重要的。 最常见的情况是需要创建一个关联 一个实体和另一个,现有的实体。

例3.3。 获得一个实体引用的例子没有初始化它的数据

```
Book book = new Book();
book.setAuthor( session.byId( Author.class ).getReference( authorId ) );
```

```
Book book = new Book();
book.setAuthor( entityManager.getReference( Author.class, authorId ) );
```

上面的工作假设实体定义允许延迟加载,一般通过 使用运行时代理。 更多信息见 吗？？？。 在这两个 情况下,将会抛出一个异常后如果给定的实体并不指实际数据库状态如果和 当应用程序试图使用返回的代理在任何方式,需要访问数据。

3.4。 获得一个实体与它的数据初始化

它也是很常见的,想获得一个实体以及与它的数据,显示为例。

例3.4。 获得一个实体引用的例子与它的数据初始化

```
session.byId( Author.class ).load( authorId );
```

```
entityManager.find( Author.class, authorId );
```

在这两种情况下将返回null如果没有匹配数据库行被发现。

3.5。 获得一个实体通过自然id

除了允许加载由标识符,Hibernate允许应用程序负载的宣布 自然的标识符。

例3.5。 简单的例子自然id访问

```
@Entity
public class User {
        @Id
        @GeneratedValue
        Long id;
```

```
        @NaturalId
        String userName;

        ...
}

// use getReference() to create associations...
Resource aResource = (Resource) session.byId( Resource.class ).getReference( 123 );
User aUser = (User) session.bySimpleNaturalId( User.class ).getReference( "steve" );
aResource.assignTo( user );


// use load() to pull initialzed data
return session.bySimpleNaturalId( User.class ).load( "steve" );
```

例3.6。 例子,自然id访问

```
import java.lang.String;

@Entity
public class User {
        @Id
        @GeneratedValue
        Long id;

        @NaturalId
        String system;

        @NaturalId
        String userName;

        ...
}

// use getReference() to create associations...
Resource aResource = (Resource) session.byId( Resource.class ).getReference( 123 );
User aUser = (User) session.byNaturalId( User.class )
                .using( "system", "prod" )
                .using( "userName", "steve" )
                .getReference();
aResource.assignTo( user );


// use load() to pull initialzed data
return session.byNaturalId( User.class )
                .using( "system", "prod" )
                .using( "userName", "steve" )
                .load();
```

就像我们在上面看到的,访问实体数据的同时自然id允许 负载 和 getReference 形式,具有相同的语义。

访问持久数据的标识符和自然id是一致的在Hibernate API。 每个定义 相同的2数据访问方法:

getReference

应该用于标识符的情况下假定存在,不存在会吗 一个实际的错误。 永远不应该被用来测试的存在。 这是因为这种方法将更喜欢创建并返回一个代理如果数据不是已经与会话关联起来 而不是去访问数据库。 典型的用例使用该方法来创建 基于外键关联。

负载

将返回持久数据关联于指定标识符值或null如果 标识符是不存在的。

除了这两个方法,每个还定义了方法 与 接受 一个 org.hibernate.LockOptions 参数。 锁定在一个单独的讨论 一章。

3.6。 刷新实体状态

你可以重新加载一个实体实例,它在任何时间的集合。

例3.7。 清爽的例子实体状态

```
Cat cat = session.get( Cat.class, catId );
...
session.refresh( cat );
```

```
Cat cat = entityManager.find( Cat.class, catId );
...
entityManager.refresh( cat );
```

在一种情况下这是有用的是当众所周知,数据库状态已经改变了自数据 读。 刷新当前数据库允许国家拖进实体实例和 持久化上下文。

另一个案例,这可能是有用的是当数据库触发器用于初始化一些 实体的属性。 注意,只有实体实例及其集合是刷新除非你 指定刷新 作为一个级联风格的任何关联。 然而,请注意 Hibernate有能力处理这种自动通过其概念生成的属性。 看到 吗？？？ 对于信息。

3.7。 修改管理/持续状态

实体管理/持续状态可能被应用程序和任何的变化 自动检测并坚持当持久化上下文是脸红。 不需要调用一个 特定的方法来让你的修改持久。

例3.8。 的例子,修改管理状态

```
Cat cat = session.get( Cat.class, catId );
cat.setName( "Garfield" );
session.flush(); // generally this is not explicitly needed
```

```
Cat cat = entityManager.find( Cat.class, catId );
cat.setName( "Garfield" );
entityManager.flush(); // generally this is not explicitly needed
```

3.8。 处理分离数据

超然是处理数据的过程范围外的任何持久化上下文。 数据变得 分离的在很多方面。 一旦持久化上下文是关闭,所有相关联的数据 变得超然。 清理持久化上下文有同样的效果。 驱逐一个特定的实体 从持久化上下文使其分离。 最后,序列化将使反序列化形式 被分离的(原来的实例仍然成功)。

分离数据仍然可以被操纵,然而持久性上下文将不再自动知道 关于这些修改和应用程序将需要进行干预,使变化持久。

3 8 1。 接续分离数据

回贴是一个过程,以一个传入的实体实例,在分离的状态 ,并与现有稀土将持久化上下文。

> **重要**
>
> JPA不提供这个模型。 这是只能通过Hibernate org hibernate会话 。

例3.9。 凶手的例子独立的实体

```
session.saveOrUpdate( someDetachedCat );
```

方法名 更新 这里有点误导。 这并不意味着一个 SQL 更新 是立即执行。 然而,它确实意味着 一个 SQL 更新 会时进行持久化上下文是吗 因为Hibernate不知道刷新它之前的状态变化来比较。 除非 实体映射与 - before - update ,在这种情况下,Hibernate将 把从数据库当前状态,看看是否需要更新。

提供实体分离, 更新 和 saveOrUpdate 操作完全相同。

3 8 2。 合并分离数据

合并的过程就是把传入的实体实例,在分离状态和复制它 数据到一个新实例,在管理国家。

例3.10。 可视化合并

```
Object detached = ...;
Object managed = entityManager.find( detached.getClass(), detached.getId() );
managed.setXyz( detached.getXyz() );
...
return managed;
```

这并非完全是发生了什么,但是其良好的可视化。

例3.11。 独立的实体合并的例子

```
Cat theManagedInstance = session.merge( someDetachedCat );
```

```
Cat theManagedInstance = entityManager.merge( someDetachedCat );
```

3.9。 检查持续状态

一个应用程序可以验证状态的实体和集合与持久化上下文。

例3.12。 例子验证管理状态

```
assert session.contains( cat );
```

```
assert entityManager.contains( cat );
```

例3.13。 懒惰的例子验证

```
if ( Hibernate.isInitialized( customer.getAddress() ) {
    //display address if loaded
}
if ( Hibernate.isInitialized( customer.getOrders()) ) ) {
    //display orders if loaded
}
if (Hibernate.isPropertyInitialized( customer, "detailedBio" ) ) {
    //display property detailedBio if loaded
}
```

```
javax.persistence.PersistenceUnitUtil jpaUtil = entityManager.getEntityManagerFactory().getPersistenceUnitU
if ( jpaUtil.isLoaded( customer.getAddress() ) {
    //display address if loaded
}
if ( jpaUtil.isLoaded( customer.getOrders()) ) ) {
    //display orders if loaded
}
if (jpaUtil.isLoaded( customer, "detailedBio" ) ) {
    //display property detailedBio if loaded
}
```

在JPA有一种替代方式来检查懒惰使用以下 javax.persistence.PersistenceUtil 模式。 然而, javax.persistence.PersistenceUnitUtil 是永远不可能的地方推荐吗

例3.14。 选择JPA意味着验证懒惰

```
javax.persistence.PersistenceUtil jpaUtil = javax.persistence.Persistence.getPersistenceUtil();
if ( jpaUtil.isLoaded( customer.getAddress() ) {
    //display address if loaded
}
if ( jpaUtil.isLoaded( customer.getOrders()) ) ) {
    //display orders if loaded
}
if (jpaUtil.isLoaded(customer, "detailedBio") ) {
    //display property detailedBio if loaded
}
```

3.10。 Hibernate api访问从JPA

JPA定义了一个令人难以置信的有用方法允许应用程序访问api的底层的提供者。

例3.15。 使用entitymanager打开

```
Session session = entityManager.unwrap( Session.class );
SessionImplementor sessionImplementor = entityManager.unwrap( SessionImplementor.class );
```

第四章。 批处理

表的内容

下面的例子展示了一个反模式为批量插入。

例4.1。 天真的方式与Hibernate插入100000行

```
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();
```

这个操作失败并且异常 OutOfMemoryException 在大约50000行大多数 系统。 原因是Hibernate缓存实例的所有新插入的客户在会话级 缓存。 有几种方法可以避免这个问题。

在批处理,使JDBC批处理。 启用JDBC批处理,设置属性 hibernate jdbc批处理大小 到一个整数10至50。

> **注意**
>
> Hibernate禁用插入配料在JDBC级透明地如果你使用一个身份标识符生成器。

如果上面的方法不合适,您可以禁用二级缓存,通过设置 hibernate缓存使用二级缓存 到 假 。

4.1。 批量插入

当你让新对象持久,使用方法 flush() 和 clear() 定期的会议,来控制一级缓存的大小。

例4.2。 冲洗和清理 会话

```
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();
```

4.2。 批量更新

当你retriev和更新数据, flush() 和 clear() 这个 定期会议。 此外,使用方法 滚动() 利用服务器端 游标为查询,该查询返回多行数据。

例4.3。使用 滚动()

```
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();
```

StatelessSession 4.3。

StatelessSession 是一个命令导向的API提供了通过Hibernate。 用它来流 数据和从数据库分离对象的形式。 一个StatelessSession 没有与之关联的持久化上下文,不提供许多高级生命周期 语义。 一些事情不提供的 StatelessSession 包括:

没有提供的特性和行为 StatelessSession

- 一级缓存
- 交互的二级或查询缓存
- 事务写入后或自动脏检查

的局限性 StatelessSession

- 操作使用无状态会话从来没有级联到关联的实例。
- 集合是忽略了一个无状态会话。
- 操作通过一个无状态会话旁路Hibernate的事件模型和拦截器。
- 由于缺乏一个一级缓存、无状态会话容易受到数据混淆效应。
- 一个无状态会话是一个低级别的抽象,是更接近底层JDBC。

例4.4。 使用 StatelessSession

```
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();
```

这个 客户 实例查询返回立即分离。 他们从不 与任何持久化上下文相关联。

这个 插入() , Update() ,和 删除() 操作定义的 StatelessSession 界面上直接操作数据库 行。 他们导致对应的SQL操作立即执行。 他们有不同的语义 这个 save() , saveOrUpdate() ,和 删除() 操作定义的 会话 接口。

4.4。 Hibernate查询语言DML

DML、或 数据标记语言 ,指的是SQL语句如 插入 , 更新 ,和 删除 。 Hibernate提供了方法DML sql风格的大部分 语句执行的形式 Hibernate查询语言(HQL) 。

4.1.1。 HQL的更新和删除

例4.5。 Psuedo-syntax更新和删除语句使用HQL

```
( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?
```

这个 吗? 后缀适应症的一个可选参数。 这个 从 和 在 条款都是可选的。

这个 从 条款只能引用一个单一的实体,它可以有别名。 如果实体名称 是别名,任何属性引用必须合格使用该别名。 如果实体的名字不是别名,然后 它是违法的任何属性引用是合格的。

连接,隐式或显式,严禁在散装HQL查询。 您可以使用子查询的 在 条款,和子查询本身可以包含连接。

例4.6。 执行一个HQL更新使用 Query.executeUpdate() 方法

```
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 = session.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

为了与EJB3规范,HQL更新语句,默认情况下,不影响版本或 时间戳属性值受影响的实体。 您可以使用一个版本更新迫使Hibernate重置 版本或时间戳属性值,通过添加 版本 关键字后 更新 关键字。

例4.7。 更新版本的时间戳

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName
int updatedEntities = session.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

> **注意**
>
> 如果你使用 版本 声明中,您不能使用自定义版本类型,使用类org.hibernate.usertype.UserVersionType 。

例4.8。 HQL 删除 声明

```
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 = session.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

方法 Query.executeUpdate() 返回一个 int 值,这表示 数量的实体操作的影响。 这可能会或可能不会相互关联的行数的影响数据库。 一个HQL批量操作可能导致多个SQL语句被执行,例如 加入了子类。 在示例的加入子类,一个 删除 反对的一个子类实际上可能导致删除表中潜在的连接,或进一步沿着继承层次结构。

10/24/11。 HQL语法插入

例4.9。 伪语法为INSERT语句

```
INSERT INTO EntityName properties_list select_statement
```

只有 插入…… 选择…… 形式的支持。 你不能指定显式值 插入。

这个 **属性列表** 类似于列规范在吗 SQL 插入 语句。 对于实体参与绘制继承,你只能使用属性直接 类级别上定义的,鉴于在**属性列表** 。 超类属性是 不允许和子类的属性是无关紧要的。 换句话说, 插入 语句 固有的非多态。

这个 **select语句** 可以是任何有效的HQL选择查询,但返回类型必须吗 匹配类型所期望的插入。 Hibernate验证返回类型在查询编译,而不是 期待该数据库检查它。 问题可能源于Hibernate类型这是等价的,而不是 平等的。 这样的一个例子是一个不匹配属性定义为一个 org.hibernate.type.DateType 和一个属性定义为一个 org.hibernate.type.TimestampType ,即使数据库可能不 做一个区别,或可能是能够处理转换。

如果 **ID** 属性中没有指定 **属性列表** , Hibernate生成一个值自动。 自动生成只提供如果你使用ID发电机这 操作数据库。 否则,Hibernate在解析过程中抛出一个异常。 可用的数据库内 发电机 org.hibernate.id.SequenceGenerator 和它的子类,和对象 实现 org.hibernate.id.PostInsertIdentifierGenerator 。 最值得注意的 异常 org.hibernate.id.TableHiLoGenerator ,这并不使一个可选择的方法 得到它的值。

对于属性映射为要么版本或时间戳,insert语句给你两个选择。 你可以 指定属性在属性列表中,在这种情况下,它的价值来自于相应的选择 表情,或省略的属性列表,在这种情况下,定义的种子值 org hibernate类型。 VersionType使用。

例4.10。 HQL INSERT语句

```
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 = session.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

11/17/11。 更多的信息在HQL

本节只简要概述HQL。 有关更多信息,请参见 吗? ? ? 。

第五章。锁定

表的内容

锁是指采取行动防止数据在关系数据库之间的变化时间阅读 和时间,使用它。

你可以锁定策略 乐观 或 悲观 。

锁定策略

乐观

乐观锁定假设多个交易可以完成而不影响对方, 因此交易能顺利进行锁定数据资源,他们影响。 在提交之前, 每笔交易确认没有其他事务修改其数据。 如果检查揭示了冲突 修改,提交事务回滚[1] 。

悲观

悲观锁定假设并发事务会相互冲突,和需要的资源 被锁在他们读,只有解锁完成后的应用程序使用的数据。

Hibernate提供了机制用于实现两种类型的锁定在你的应用程序。

5.1。乐观

当你的应用程序使用长事务或对话,跨几个数据库事务,你可以 商店版本数据,所以,如果相同的实体是更新两个对话,最后提交变更是 通知的冲突,不覆盖其他对话的作品。 这种方法保证一些 隔离,但良好的可扩展性和作品的特别好 经常阅读写有时 情况。

Hibernate提供了两种不同的机制来存储版本信息,一个专用的版本号或一个 时间戳。

版本号

时间戳

> **注意**
>
> 一个版本或时间戳属性不能为空一个超然的实例。 Hibernate可以检测到任何实例用 空版本或时间戳作为瞬态,无论其他未保存的值策略,您指定。 宣布 一个可以为空的版本或时间戳属性是一个简单的方法来避免问题的传递回贴 Hibernate,特别有用如果你使用指定的标识符或组合键。

5.1.1。 专用版本号

版本号机制提供了通过一个乐观锁 version 注释。

例5.1。 @ version注释

```
@Entity
public class Flight implements Serializable {
...
    @Version
    @Column(name="OPTLOCK")
    public Integer getVersion() { ... }
```

```
}
```

在这里,版本属性映射到 OPTLOCK 列,以及使用它的实体管理器 来检测冲突的更新和防止损失的更新,将被覆盖的 最后提交的胜 策略。

版本字段可以是任何类型的类型,只要你定义和实施适当的 UserVersionType 。

你的应用程序是禁止改变版本号设置的冬眠。 人为地增加 版本号,请参阅文档属性 LockModeType.OPTIMISTIC_FORCE_INCREMENT 或 LockModeType.PESSIMISTIC_FORCE_INCREMENTcheck 在 Hibernate实体管理器参考 文档。

> **数据库生成的版本号**
>
> 如果版本号是由数据库生成,如一个触发器,使用注释 @org.hibernate.annotations.Generated(GenerationTime.ALWAYS) 。

例5.2。 声明一个版本属性 hbm xml

```
<version
        column="version_column"
        name="propertyName"
        type="typename"
        access="field|property|ClassName"
        unsaved-value="null|negative|undefined"
        generated="never|always"
        insert="true|false"
        node="element-name|@attribute-name|element/@attribute|."
/>
```

列名	列的名称控股版本号。 可选的,默认属性 的名字。
类型	属性的名称的持久化类。
访问	这个类型的版本号。 可选的,默认为 整数 。
未保存的价值	Hibernate的策略来访问属性值。 可选的,默认为 财产 。
生成	表明新实例化一个实例,因此未保存的。 这个区别它 从分离实例被保存或装载在之前的会话。 默认值,未定义 ,表明该标识符属性值应该是 使用。 可选的。
插入	表明该版本属性值是由数据库生成。 可选的,默认值 到 从来没有 。
	是否包括 版本 列在SQL 插入 语句。 默认为 真正的 ,但是你可以设置它 假 如果 数据库列定义了一个默认值 0 。

5.1.2中所述。时间戳

时间戳是一个不可靠的乐观锁定方式比版本号,但是可以使用的应用程序 对于其他用途。 如果你使用时间戳是自动的 version 注释 日期 或 日历 。

例5.3。 使用乐观锁定的时间戳

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

Hibernate可以检索时间戳值从数据库或JVM,通过阅读您指定的值为 这个 @org.hibernate.annotations.Source 注释。 值可以是 org.hibernate.annotations.SourceType.DB 或 org.hibernate.annotations.SourceType.VM 。 默认行为是使用数据库,是 还使用如果你不指定注释在所有。

时间戳也可以由数据库生成而不是冬眠,如果你使用 @org.hibernate.annotations.Generated(GenerationTime.ALWAYS) 注释。

例5.4。 时间戳元素 hbm xml

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

列名	列的名称,持有这些时间戳。 可选的,默认属性 namel
	javabean样式的名称属性的Java类型日期或时间戳的持久性 类。

称访问	Hibernate使用的策略来访问属性值。 可选的,默认为 财产 。
未保存的价值	这表明一个版本属性比实例是新 实例化,未保存的。 这有别于分离实例被保存或装载在一个 之前的会话。 默认值未定义 表明Hibernate使用 标识符属性值。
源	冬眠的时间戳是否从数据库检索或电流 JVM。 基于数据库时间戳招致额外的开销,因为Hibernate需要查询数据库每个时间 确定下一个值的增量。 然而,数据库导出时间戳是更安全的使用 集群环境中。 并不是所有的数据库方言已知支持数据库的检索 当前的时间戳。 别人也可能不安全的锁定,因为缺乏精度。
生成	无论时间戳属性值是由数据库生成。 可选的,默认为 从来没有 。

5.2。悲观

通常,您只需要指定一个隔离级别的JDBC连接,让数据库处理 锁定问题。 如果你需要获得独家悲观锁或重新获得锁在开始一个新的 事务,Hibernate提供了必要的工具。

> **注意**
>
> Hibernate总是使用数据库的锁定机制,和从来没有锁对象在内存中。

5 2 1。 LockMode 类

这个 LockMode 类定义了不同的锁,Hibernate可以获得水平。

LockMode.WRITE	收购时自动更新或插入一个行冬眠。
LockMode.UPGRADE	获得明确的用户请求时使用 选择…… 更新 在数据库 支持,语法。
LockMode.UPGRADE_NOWAIT	获得明确的用户请求时使用 选择…… 对于更新NOWAIT 在 甲骨文。
LockMode.READ	读取数据时自动获得下冬眠 可重复读 或 可序列化的 隔离级别。 它可以重新获得用户明确 请求。
LockMode.NONE	没有一个锁。 所有对象锁定模式切换到该结束的时候 事务。 与会话关联的对象通过调用 Update() 或 saveOrUpdate() 也开始在这个锁定模式。

明确用户的要求上面提到的发生由于下列行动:

- 一个叫 会话负荷() ,指定一个LockMode。
- 一个叫 会话锁() 。
- 一个叫 Query.setLockMode() 。

如果你打电话 会话负荷() 对于选项 **升级** 或 **UPGRADE_NOWAIT** ,请求的对象不是已经加载的会话,对象 加载使用选择…… 更新 。 如果你打电话 load() 为一个对象, 已经装满一个限制较少的锁比你请求,Hibernate电话吗 锁() 为该对象。

会话锁() 执行版本号检查如果指定的锁模式 读 , 升级 ,或 UPGRADE_NOWAIT 。 对于 升级 或 UPGRADE_NOWAIT , 选择…… 更新 语法是 使用。

如果所请求的锁模式不支持数据库,Hibernate使用一个适当的替代模式 而不是抛出异常。 这可以确保应用程序是可移植的。

[1] http://en.wikipedia.org/wiki/Optimistic_locking

第六章。缓存

表的内容

6.1。 查询缓存

如果你有查询,反复运行,同样的参数,查询缓存提供的性能收益。

缓存介绍领域的开销事务处理。 例如,如果您的缓存查询的结果 针对一个对象,Hibernate需要记录是否已经提交任何变化对对象, 和相应的缓存失效。 此外,受益于缓存查询结果是有限的,和高度 取决于您的应用程序的使用模式。 由于这些原因,Hibernate查询缓存的禁用 默认的。

程序6.1。 启用查询缓存

1. **设置 hibernate缓存使用查询缓存 财产 真正的 。**

   这个设置创建两个新的缓存区域:

   - org.hibernate.cache.internal.StandardQueryCache 保存缓存的查询结果。
   - org.hibernate.cache.spi.UpdateTimestampsCache 持有时间戳的最新更新 可查询的表。 这些时间戳验证结果的查询缓存服务。

2. **调整缓存超时底层的缓存区域**

   如果你配置你的潜在的缓存实现使用期满或超时,超时的设置缓存 底层的缓存区域 UpdateTimestampsCache 到一个更高的价值比超时的 查询缓存。 它是可能的,并建议,设置UpdateTimestampsCache地区永远 到期。 具体而言,一个LRU(最近最少使用)缓存过期的政策是不合适的。

3. **使结果缓存为特定的查询**

   由于大多数查询不受益于缓存的结果,您需要启用缓存为个人 查询,查询缓存启用了e ven整体。 使结果缓存为特定的查询、调用 org.hibernate.Query.setCacheable(真正的) 。 这叫允许查询来寻找 现有的缓存结果或添加到缓存的结果当它被执行。

查询缓存不缓存实际的实体的状态在缓存中。 它缓存标识符值和 值类型的结果。 因此,总是使用查询缓存结合二级 缓存对于那些实体应该缓存作为查询结果缓存。

但是。 查询缓存区域

对细粒度控制查询缓存过期策略,指定一个命名为一个特定的缓存区域 查询通过调用 Query.setCacheRegion() 。

例6.1。方法 setCacheRegion

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
        .setEntity("blogger", blogger)
        .setMaxResults(15)
        .setCacheable(true)
        .setCacheRegion("frontpages")
        .list();
```

强制查询缓存刷新它的一个地区,无视任何缓存的结果在该地区,叫 org.hibernate.Query.setCacheMode(CacheMode.REFRESH) 。 结合该地区的定义 给定的查询,Hibernate缓存刷新结果选择性地在特定的地区。 这是更 效率比散装驱逐的地区通过 org.hibernate.SessionFactory.evictQueries() 。

6.2。 二级缓存提供者

Hibernate是兼容与几个二级缓存提供者。 没有一个供应商的支持 Hibernate的可能的缓存策略。 节6 2 3,“第二级缓存提供者Hibernate” 列出供应商,以及 他们的接口和支持缓存策略。 定义的缓存策略,看到 6.2.2节,“缓存策略” 。

6 2 1。 配置缓存提供者

您可以配置您的缓存提供者或者使用注释或映射文件。

实体。 默认情况下,实体不是部分的二级缓存,不推荐使用。 如果你 绝对必须使用实体,设置 共享缓存模式 元素 **persistence . xml** ,或用财产 javax.persistence.sharedCache.mode 在你的配置。 使用其中一个值 表格6.1,“可能值共享缓存模式” 。

表6.1。 可能值共享缓存模式

价值	描述
使选择性	实体不缓存,除非你明确其标记为可缓存的。 这是默认的和 推荐值。
禁用选择性	实体是缓存的除非你明确标明不能缓存。
所有	所有实体总是缓存即使你标明不能缓存。
没有	没有实体缓存即使你马克他们作为缓存。 这个选项主要禁用二级 缓存。

设置全局默认缓存并发策略缓存并发策略的 hibernate缓存默认缓存并发策略 配置属性。 看到 6.2.2节,“缓存策略” 对于可能的值。

> **注意**

在可能的情况下,定义缓存并发策略而不是全球每个实体。 使用 @org.hibernate.annotations.Cache 注释。

例6.2。 配置缓存提供者使用注释

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
```

你可以缓存的内容收集或标识符,如果集合包含其他实体。 使用 这个 @Cache 注释在集合属性。

@Cache 可以带一些属性。

属性的 @Cache 注释

使用

给定的缓存并发性战略,可能是:

- 没有
- READ_ONLY
- NONSTRICT_READ_WRITE
- READ_WRITE
- 事务性

地区

缓存区。 这个属性是可选的,默认为完全限定类名的 类,或qually-qualified角色的集合的名称。

包括

是否包括所有的属性. . 可选的,可采取两种可能的值。

- 一个值的 所有 包括所有的属性。 这是默认的。
- 一个值的 非延迟 只包括非延迟特性。

例6.3。 配置缓存提供者使用映射文件

```
<cache
   usage="transactional"
   region="RegionName"
   include="all"
/>
```

就像在 例6.2, “配置缓存提供者使用注释” ,你可以提供属性 映射文件。 有一些特定的语法差异属性映射文件。

使用

缓存策略。 这个属性是必需的,并且可以是任何下列值。

- 事务性
- 读写
- nonstrict-read-write
- 只读

地区

二级缓存的名称区域。 这个可选属性默认为类或集合 角色名。

包括

实体的属性映射是否与 懒= true 可以缓存当 必须启用惰性抓取。 默认为 所有 也可以 非延迟 。

而不是 <缓存> ,你可以使用 <类缓存> 和 <收集缓存> 元素 **hibernate cfg xml** 。

6.2.2。 缓存策略

只读

一个只读缓存有利于数据,需要经常读取,但不能修改。 它很简单,执行 嗯,和使用是安全的,在集群环境中。

nonstrict读写

一些应用程序只有很少需要修改数据。 这样如果两个交易不太可能 尝试更新同一项目同时。 在这种情况下,您不需要严格的事务隔离, 和一个nonstrict-read-write缓存可能是适当的。 如果缓存中使用JTA的环境,你必须 指定 manager_lookup_class 。 在其他环境中,ensore 事务完成前你打电话 会话关闭() 或 会话断开() 。

读写

一个读写缓存是适合一个应用程序需要定期更新数据。 不要使用 读写策略如果你需要序列化事务隔离。 在一个JTA环境,指定一个 策略获取JTA TransactionManager通过设置属性 manager_lookup_class 。 在非jta环境,确保 事务完成前你打电话 会话关闭() 或 会话断开() 。

> **注意**
>
> 使用读写策略在集群环境中,底层的缓存实现必须 支持锁定。 这个内置缓存提供者不支持锁定。

事务性

事务缓存策略提供支持事务缓存提供者如JBoss TreeCache。 你只能使用这样一个缓存在JTA的环境,你必须首先指定 manager_lookup_class 。

6 2 3。 Hibernate二级缓存提供者

缓存	接口	支持策略
哈希表(测试只有)		• 只读 • nontrict读写 • 读写
EHCache		• 只读 • nontrict读写 • 读写 • 事务性
Infinispan		• 只读 • 事务性

6.3。 管理缓存

6 3 1。 移动物品的缓存

操作,添加一个条目到内部缓存的会话

保存或更新一个项目

- save()
- Update()
- saveOrUpdate()

检索项

- load()
- get()
- 列表()
- 迭代()
- 滚动()

同步或删除缓存项。 对象的状态与数据库同步,当你调用方法 flush() 。 为了避免这种同步,您可以删除对象和所有的收藏品从第一级缓存的 驱逐() 法。 删除所有物品的 会话缓存,使用方法 会话清楚() 。

例6.4。 将一个项目从一级缓存

```
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);
}
```

决定一个项目是否属于会话缓存。 会话提供了一个 包含() 方法来确定一个实例属于 会话缓存。

例6.5。 二级缓存回收

你可以驱逐缓存实例的状态,整个类,收集实例或整个集合的作用, 使用方法 SessionFactory 。

```
sessionFactory.getCache().containsEntity(Cat.class, catId); // is this particular Cat currently in the cache

sessionFactory.getCache().evictEntity(Cat.class, catId); // evict a particular Cat

sessionFactory.getCache().evictEntityRegion(Cat.class);  // evict all Cats
```

```
sessionFactory.getCache().evictEntityRegions();  // evict all entity data
sessionFactory.getCache().containsCollection("Cat.kittens", catId); // is this particular collection currently in t
sessionFactory.getCache().evictCollection("Cat.kittens", catId); // evict a particular collection of kittens
sessionFactory.getCache().evictCollectionRegion("Cat.kittens"); // evict all kitten collections
sessionFactory.getCache().evictCollectionRegions(); // evict all collection data
```

6 3 1 1。 一个会话之间的交互和二级缓存

这个CacheMode控制一个特定的会话与第二级缓存。

CacheMode.NORMAL	读取和写入项目的二级缓存。
CacheMode.GET	从第二级缓存读取项目,但不写入二级缓存除了 更新数据。
CacheMode.PUT	写项目的二级缓存。 它不读从二级缓存。 它绕过 的影响 hibernate缓存使用最小的把 和部队一个刷新的 二级缓存为所有项目从数据库读取。

6 3 1 2。 浏览内容的第二级或查询缓存区域

启用统计信息后,您可以浏览内容的二级缓存或查询缓存区域。

程序6.2。 启用统计信息

1. 集 hibernate生成统计 到 真正的 。
2. 可选地,设置 hibernate缓存使用结构化的条目 到 真正的 ,导致 Hibernate缓存条目存储在一个人类可读的格式。

例6.6。 浏览二级缓存条目通过统计数据API

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

第七章。服务

表的内容

7.1。 什么是服务?

服务类和可插入的实现提供Hibernate各种类型的 功能。 特别是他们实现某些服务契约接口。 接口 被称为服务角色;实现类是知道作为服务实现。 一般 来说,用户可以插入交替实现的标准服务角色(覆盖);他们可以 还定义额外的服务超越基础组服务角色(扩展)。

7.2。 服务合同

的基本要求是实现服务的标记接口 org hibernate服务服务 。 Hibernate使用这个内部对于一些 基本类型安全。

可选地,该服务还可以实现 org hibernate服务spi可启动 和 org hibernate服务spi可停止的 接口来接收通知 开始和停止的。 另一个可选的服务合同 org hibernate服务spi可控的 这标志着服务为可控的 在JMX提供了JMX集成是启用的。

7.3。 服务依赖关系

服务是允许声明依赖于其他服务可以使用2种方式。

7.3.1。@ org.hibernate.service.spi.InjectService

任何方法在服务实现类接受单个参数和标注 @ InjectService 被认为是另一个服务请求注入。

默认方法参数的类型将是服务的角色被注入。 如果 参数类型是不一样的服务角色, serviceRole 属性 的 InjectService 应该用于显式命名的角色。

默认情况下注射服务被认为是必需的,这是启动将会失败如果命名 依赖服务缺失。 如果该服务被注入是可选的, 需要 属性的 InjectService 应该声明为 假 (默认是 真正的)。

7 3 2。 org.hibernate.service.spi.ServiceRegistryAwareService

第二种方法是一个拉的方法,服务实现了可选的服务接口 org.hibernate.service.spi.ServiceRegistryAwareService 声明 一个 injectServices 法。 在启动时,Hibernate将注入 org.hibernate.service.ServiceRegistry 本身成服务, 实现这个接口。 服务可以使用 ServiceRegistry 参考定位任何额外的服务需要。

ServiceRegistry 7.4。

中央服务API,除了服务本身,是 org.hibernate.service.ServiceRegistry 接口。 的主要目的 服务注册中心是持有、管理和提供服务。

服务注册中心是分等级的。 服务在一个注册表可以依赖和利用服务 相同的注册表以及任何母公司注册。

使用 org.hibernate.service.ServiceRegistryBuilder 建立一个 org.hibernate.service.ServiceRegistry 实例。

7.5。 标准服务

7.5.1。 org.hibernate.engine.jdbc.batch.spi.BatchBuilder

笔记

定义了策略管理JDBC语句Hibernate如何配料

引发剂

org.hibernate.engine.jdbc.batch.internal.BatchBuilderInitiator

实现

org.hibernate.engine.jdbc.batch.internal.BatchBuilderImpl

7.5.2。 org.hibernate.service.config.spi.ConfigurationService

笔记

提供了访问配置设置,结合这些显式也提供 作为这些贡献的任何注册 org hibernate积分器spi积分器 实现

引发剂

org.hibernate.service.config.internal.ConfigurationServiceInitiator

实现

org.hibernate.service.config.internal.ConfigurationServiceImpl

7 5 3。 org.hibernate.service.jdbc.connections.spi.ConnectionProvider

笔记

定义意味着,Hibernate可以获得和释放 java sql连接 对它的使用实例。

引发剂

org.hibernate.service.jdbc.connections.internal.ConnectionProviderInitiator

实现

- org.hibernate.service.jdbc.connections.internal.C3P0ConnectionProvider —— 基于集成提供了连接池与C3P0连接池的图书馆
- org.hibernate.service.jdbc.connections.internal.DatasourceConnectionProviderImpl —— 提供连接管理委托给一个 javax.sql.DataSource
- org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl —— 提供基本的连接池基于简单的自定义池。 注意目的 生产使用!
- org.hibernate.service.jdbc.connections.internal.ProxoolConnectionProvider —— 基于集成提供了连接池的连接池proxool图书馆
- org.hibernate.service.jdbc.connections.internal.UserSuppliedConnectionProviderImpl —— 没有提供连接的支持。 显示用户将提供连接直接冬眠。 不建议使用。

7 5 4。 org.hibernate.service.jdbc.dialect.spi.DialectFactory

笔记

合同对Hibernate来获得 org hibernate方言方言 实例使用。 这要么是明确定义的 hibernate方言 财产或决定的 部分7.5.5,“ org.hibernate.service.jdbc.dialect.spi.DialectResolver ”服务,这是一种委托给该服务。

引发剂

org.hibernate.service.jdbc.dialect.internal.DialectFactoryInitiator

实现

org.hibernate.service.jdbc.dialect.internal.DialectFactoryImpl

7.5.5。 org.hibernate.service.jdbc.dialect.spi.DialectResolver

笔记

提供解决 org hibernate方言方言 使用基于 从元数据中提取信息的JDBC。

标准的解析器实现充当一个链,委托给一系列的个人 解析器。 标准的Hibernate决议的行为都包含在 org.hibernate.service.jdbc.dialect.internal.StandardDialectResolver 。 org.hibernate.service.jdbc.dialect.internal.DialectResolverInitiator 还担任了 hibernate方言解析器 设置任何 自定义解析器。

引发剂

org.hibernate.service.jdbc.dialect.internal.DialectResolverInitiator

实现

org.hibernate.service.jdbc.dialect.internal.DialectResolverSet

7 5 6。 org.hibernate.engine.jdbc.spi.JdbcServices

笔记

特殊类型的服务,聚集在一起一些其它的服务和提供 一个高级设置功能。

引发剂

org.hibernate.engine.jdbc.internal.JdbcServicesInitiator

实现

org.hibernate.engine.jdbc.internal.JdbcServicesImpl

7 5 7。 org.hibernate.service.jmx.spi.JmxService

笔记

提供简化访问JMX相关所需的功能,Hibernate。

引发剂

org.hibernate.service.jmx.internal.JmxServiceInitiator

实现

- org.hibernate.service.jmx.internal.DisabledJmxServiceImpl —— 一个空操作实现功能被禁用时JMX。
- org.hibernate.service.jmx.internal.JmxServiceImpl —— 标准实施处理JMX

7 5 8。 org.hibernate.service.jndi.spi.JndiService

笔记

提供简化访问JNDI相关所需的功能,Hibernate。

引发剂

org.hibernate.service.jndi.internal.JndiServiceInitiator

实现

org.hibernate.service.jndi.internal.JndiServiceImpl

7 5 9。 org.hibernate.service.jta.platform.spi.JtaPlatform

笔记

提供了一个抽象从底层JTA平台当使用JTA特性。

引发剂

org.hibernate.service.jta.platform.internal.JtaPlatformInitiator

> **重要**
>
> JtaPlatformInitiator 提供了映射到遗留, 现在弃用 org.hibernate.transaction.TransactionManagerLookup 名称为hibernate提供内部 org.hibernate.transaction.TransactionManagerLookup 实现。

实现

- org.hibernate.service.jta.platform.internal.BitronixJtaPlatform —— 集成与Bitronix独立的事务管理器。
- org.hibernate.service.jta.platform.internal.BorlandEnterpriseServerJtaPlatform —— 综合事务管理器作为部署在一个宝蓝企业服务器
- org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform —— 综合事务管理器作为被部署到JBoss应用服务器
- org.hibernate.service.jta.platform.internal.JBossStandAloneJtaPlatform —— 集成与JBoss事务独立的事务管理器
- org.hibernate.service.jta.platform.internal.JOTMJtaPlatform —— 集成与JOTM独立的事务管理器
- org.hibernate.service.jta.platform.internal.JOnASJtaPlatform —— 集成和乔纳斯事务管理器。
- org.hibernate.service.jta.platform.internal.JRun4JtaPlatform —— 综合事务管理器作为部署在一个应用服务器JRun 4。
- org.hibernate.service.jta.platform.internal.NoJtaPlatform —— 无操作版本当没有JTA设置配置
- org.hibernate.service.jta.platform.internal.OC4JJtaPlatform —— 事务管理器集成为部署在一个OC4J(Oracle)应用程序 服务器。
- org.hibernate.service.jta.platform.internal.OrionJtaPlatform —— 整合与事务管理器是一个应用服务器部署在猎户座。
- org.hibernate.service.jta.platform.internal.ResinJtaPlatform —— 事务管理器集成为部署在树脂应用服务器。
- org.hibernate.service.jta.platform.internal.SunOneJtaPlatform —— 事务管理器集成为部署在一个太阳一个(7及以上) 应用程序服务器。
- org.hibernate.service.jta.platform.internal.TransactionManagerLookupBridge —— 提供一个桥梁遗留(弃用) org.hibernate.transaction.TransactionManagerLookup 实现
- org.hibernate.service.jta.platform.internal.WebSphereExtendedJtaPlatform —— 事务管理器集成为部署在一个WebSphere Application Server (6及以上)。
- org.hibernate.service.jta.platform.internal.WebSphereJtaPlatform —— 事务管理器集成为部署在一个WebSphere Application Server (4、5.0和5.1)。
- org.hibernate.service.jta.platform.internal.WeblogicJtaPlatform —— 事务管理器集成为部署在Weblogic应用程序服务器。

7 5 10。 org.hibernate.service.jdbc.connections.spi.MultiTenantConnectionProvider

笔记

一个变化的 节7 5 3。" org.hibernate.service.jdbc.connections.spi.ConnectionProvider " 提供JDBC 连接在多租户环境中。

引发剂

N / A

实现

希望用户如果需要提供适当的实现。

7 5 11。 org.hibernate.persister.spi.PersisterClassResolver

笔记

确定合适的合同 org.hibernate.persister.entity.EntityPersister 或 org.hibernate.persister.collection.CollectionPersister 实现类使用给定一个实体或集合映射。

引发剂

org.hibernate.persister.internal.PersisterClassResolverInitiator

实现

org.hibernate.persister.internal.StandardPersisterClassResolver

7 5 12。 org.hibernate.persister.spi.PersisterFactory

笔记

工厂来创建 org.hibernate.persister.entity.EntityPersister 和 org.hibernate.persister.collection.CollectionPersister 实例。

引发剂

org.hibernate.persister.internal.PersisterFactoryInitiator

实现

org.hibernate.persister.internal.PersisterFactoryImpl

7 5 13。 org.hibernate.cache.spi.RegionFactory

笔记

集成点Hibernate二级缓存支持。

引发剂

org.hibernate.cache.internal.RegionFactoryInitiator

实现

- org.hibernate.cache.ehcache.EhCacheRegionFactory
- org.hibernate.cache.infinispan.InfinispanRegionFactory
- org.hibernate.cache.infinispan.JndiInfinispanRegionFactory
- org.hibernate.cache.internal.NoCachingRegionFactory
- org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory

7 5 14。 org.hibernate.service.spi.SessionFactoryServiceRegistryFactory

笔记

工厂来创建 org.hibernate.service.spi.SessionFactoryServiceRegistry 作为一个专业的实例 org.hibernate.service.ServiceRegistry 对于 org hibernate sessionfactory 作用域服务。 看到 节7 7 2、 “SessionFactory注册表” 为更多的细节。

引发剂

org.hibernate.service.internal.SessionFactoryServiceRegistryFactoryInitiator

实现

org.hibernate.service.internal.SessionFactoryServiceRegistryFactoryImpl

7 5 15。 org hibernate统计统计

笔记

合同收集统计数据暴露。 统计数据在收集通过 org.hibernate.stat.spi.StatisticsImplementor 合同。

引发剂

org.hibernate.stat.internal.StatisticsInitiator

定义了一个 hibernate统计工厂 设置允许 配置 org.hibernate.stat.spi.StatisticsFactory 使用内部 在构建实际的 org hibernate统计统计 实例。

实现

org.hibernate.stat.internal.ConcurrentStatisticsImpl

默认 org.hibernate.stat.spi.StatisticsFactory 实现构建一个 org.hibernate.stat.internal.ConcurrentStatisticsImpl 实例。

7 5 16。 org.hibernate.engine.transaction.spi.TransactionFactory

笔记

策略定义如何冬眠的 org hibernate事务 API映射到底层事务的方法。

引发剂

org.hibernate.stat.internal.StatisticsInitiator

定义了一个 hibernate统计工厂 设置允许 配置 org.hibernate.stat.spi.StatisticsFactory 使用内部 在构建实际的 org hibernate统计统计 实例。

实现

- org.hibernate.engine.transaction.internal.jta.CMTTransactionFactory —— 一个jta的基础战略,Hibernate不控制事务。 一个 重要的区别是,交互与底层JTA的实现 是通过 javax.transaction.TransactionManager
- org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory —— 一个非jta策略中使用JDBC事务管理 java sql连接
- org.hibernate.engine.transaction.internal.jta.JtaTransactionFactory —— 一个jta的基础战略,Hibernate *可能* 是控制事务。 一个 重要的区别是,交互与底层JTA 实现通过 javax.transaction.UserTransaction

7 5 17。 org.hibernate.tool.hbm2ddl.ImportSqlCommandExtractor

笔记

合同中提取的声明 导入sql 脚本。

引发剂

org.hibernate.tool.hbm2ddl.ImportSqlCommandExtractorInitiator

实现

- org.hibernate.tool.hbm2ddl.SingleLineSqlCommandExtractor 对待每一行作为一个完整的SQL语句。 注释行应当开始 —— , / / 或 / * 字符 序列。
- org.hibernate.tool.hbm2ddl.MultipleLinesSqlCommandExtractor 支持指令/评论和引用字符串传播到多个行。每个 语句必须以分号结束结束。

7.6。 定制服务

一旦一个 org.hibernate.service.ServiceRegistry 是建立被认为是吗 不可变的;服务本身可能会接受,但不变性进行了这个意思 添加/更换服务。 所以提供的另一个角色 org.hibernate.service.ServiceRegistryBuilder 是允许调整的服务吗 这将会包含在 org.hibernate.service.ServiceRegistry 从它产生。

有两个方法来告知 org.hibernate.service.ServiceRegistryBuilder 约 定制服务。

- 实现一个 org.hibernate.service.spi.BasicServiceInitiator 类 控制服务的按需构造类并将其添加到 org.hibernate.service.ServiceRegistryBuilder 通过其 addInitiator 法。
- 就实例化服务类,并将它添加到 org.hibernate.service.ServiceRegistryBuilder 通过其 addService 法。

要么方法添加一个服务方法或添加一个启动程序的方法是有效的扩展了 注册表(添加新服务角色)和覆盖服务(替换服务实现)。

7.7。 特殊服务注册

7 7 1。 开机注册

开机注册的服务,绝对要拥有可供大多数事情的工作。 这里的主要服务是 节7 7 1 1 1”, org.hibernate.service.classloading.spi.ClassLoaderService “ 这是一个完美的例子。 甚至解决配置文件需要访问类加载服务(资源看起来ups)。 这 是根注册中心(没有父母)在正常使用。

开机注册建造的实例使用 org.hibernate.service.BootstrapServiceRegistryBuilder 类。

例7.1。 使用BootstrapServiceRegistryBuilder

```
BootstrapServiceRegistry bootstrapServiceRegistry = new BootstrapServiceRegistryBuilder()
        // pass in org.hibernate.integrator.spi.Integrator instances which are not
        // auto-discovered (for whatever reason) but which should be included
        .with( anExplicitIntegrator )
        // pass in a class-loader Hibernate should use to load application classes
        .withApplicationClassLoader( anExplicitClassLoaderForApplicationClasses )
        // pass in a class-loader Hibernate should use to load resources
        .withResourceClassLoader( anExplicitClassLoaderForResources )
        // see BootstrapServiceRegistryBuilder for rest of available methods
        ...
        // finally, build the bootstrap registry with all the above options
        .build();
```

7 7 1 1。 引导注册服务

7 7 1 1 1。 org.hibernate.service.classloading.spi.ClassLoaderService

Hibernate需要交互的类加载器。 然而,Hibernate的方式 (或任何库)应该与不同的类加载器基于运行时环境 这是托管应用程序。 应用程序服务器,OSGi容器和其他模块 类加载系统施加非常具体的类加载需求。 这个服务是提供 Hibernate抽象从这个环境的复杂性。 同样重要的是,它确实 所以在一个单一的可切换组件的方式。

从一个类加载器交互,Hibernate需要以下功能:

- 定位应用程序类的能力
- 能够找到集成类
- 定位资源的能力(属性文件、xml文件等)
- 能力负荷 java util即便serviceloader

> **注意**
>
> 目前,能够装载应用程序类和能力负荷集成 类被组合到一个“负载类”功能的服务。 这可能改变在一个版本中。

7 7 1 1 2。 org.hibernate.integrator.spi.IntegratorService

应用程序、插件和其他所有需要集成Hibernate使用要求 一些东西,通常应用程序,协调登记件每个集成 需要代表每个积分器。 该服务的目的是让那些集成商 被发现和让他们自己与Hibernate集成。

这个服务侧重于探索方面。 它使用标准的Java java util即便serviceloader 能力提供的 org.hibernate.service.classloading.spi.ClassLoaderService 为了发现的实现 org hibernate积分器spi积分器 合同。 集成商可以简单地定义一个文件命名 **/ meta - inf / services / org hibernate积分器spi积分器** 并使它 可以在类路径中。 java util即便serviceloader 涵盖了 这个文件的格式在细节,但本质上这类的列表实现。FQN org hibernate积分器spi积分器 每行一个。

看到 7.9节,“集成商”

7 7 2。 SessionFactory注册

虽然它是最佳实践来治疗所有的注册表实例类型针对给定的 org hibernate sessionfactory ,这组服务实例 明确属于一个单一的 org hibernate sessionfactory 。 这个 不同的是时间问题在当他们需要启动。 通常他们需要访问 org hibernate sessionfactory 开始。 这个特殊注册表是 org.hibernate.service.spi.SessionFactoryServiceRegistry

7 7 2 1。 org.hibernate.event.service.spi.EventListenerRegistry

笔记

服务管理事件监听器。

引发剂

org.hibernate.event.service.internal.EventListenerServiceInitiator

实现

org.hibernate.event.service.internal.EventListenerRegistryImpl

7.8。 使用服务和注册

很快……

7.9。集成商

这个 org hibernate积分器spi积分器 的目的是提供一个简单的吗 意味着允许开发者嵌入过程建立一个正常的 SessionFactory。 这个 这个 org hibernate积分器spi积分器 接口定义2的方法 兴趣: 整合 允许我们钩进建筑过程; 瓦解 允许我们钩成SessionFactory关闭。

> **注意**
>
> 有一个第三方法上定义的 org hibernate积分器spi积分器 , 一个重载的形式的 整合 接受 org.hibernate.metamodel.source.MetadataImplementor 而不是

org hibernate cfg配置 。 此表格适用于与新 元模型代码的完成时间表在5.0

看到 节7 7 1 1 2”，org.hibernate.integrator.spi.IntegratorService ”

除了发现IntegratorService提供的方法,应用程序可以手动 注册建造BootstrapServiceRegistry时积分器的实现。 看到 例7.1,“使用BootstrapServiceRegistryBuilder”

7 - 9 - 1。 积分器用例

主要的用例的 org hibernate积分器spi积分器 正确 现在正在注册事件侦听器和提供服务(见org.hibernate.integrator.spi.ServiceContributingIntegrator)。 5.0 我们计划扩大,允许改变元模型描述对象之间的映射和关系模型。

例7.2。 注册事件侦听器

```
public class MyIntegrator implements org.hibernate.integrator.spi.Integrator {

    public void integrate(
            Configuration configuration,
            SessionFactoryImplementor sessionFactory,
            SessionFactoryServiceRegistry serviceRegistry) {
        // As you might expect, an EventListenerRegistry is the thing with which event listeners are registered
        // service so we look it up using the service registry
        final EventListenerRegistry eventListenerRegistry = serviceRegistry.getService( EventListenerRegistry.cla

        // If you wish to have custom determination and handling of "duplicate" listeners, you would have to a
        // implementation of the org.hibernate.event.service.spi.DuplicationStrategy contract like this
        eventListenerRegistry.addDuplicationStrategy( myDuplicationStrategy );

        // EventListenerRegistry defines 3 ways to register listeners:
        //     1) This form overrides any existing registrations with
        eventListenerRegistry.setListeners( EventType.AUTO_FLUSH, myCompleteSetOfListeners );
        //     2) This form adds the specified listener(s) to the beginning of the listener chain
        eventListenerRegistry.prependListeners( EventType.AUTO_FLUSH, myListenersToBeCalledFirst );
        //     3) This form adds the specified listener(s) to the end of the listener chain
        eventListenerRegistry.appendListeners( EventType.AUTO_FLUSH, myListenersToBeCalledLast );
    }
}
```

第八章。 数据分类

表的内容

Hibernate既了解Java和JDBC应用程序数据的表示。 读和写的能力 对象数据到数据库被称为 编组 ,的功能是一个Hibernate类型 。 一个 类型 是一个实现的吗 org hibernate类型类型 接口。 Hibernate 类型 描述 行为的各个方面的Java类型如如何检查平等和如何克隆值。

> **这个词的用法 类型**
>
> Hibernate 类型 既不是一个Java类型还是一个SQL数据类型。 它提供的信息 这两种。 当你遇到术语 类型 对于Hibernate来说,它可能参考Java类型, JDBC类型,或Hibernate类型,这取决于上下文。

Hibernate分类类型分为两个高级组: 8.1节,“价值类型” 和 8.2节,“实体类型” 。

8.1。 值类型

一个 值类型 没有定义其自己的生命周期。 实际上,它属于一个 8.2节,“实体类型” ,它定义了它 生命周期。 值类型是进一步分为三个子类别。

- 节8 1 1”,基本类型”
- 节8 1 2,“复合类型”
- 节8 1 3、 “集合类型”

8 1 1。 基本类型

基本值类型通常映射一个数据库值,或列,一个单一的、非聚合的Java 类型。 Hibernate提供了一些内置的基本类型,按照自然的映射的推荐 JDBC规范。 你可以覆盖这些映射,并提供和使用替代映射。 这些主题 进一步讨论。

表8.1。 基本类型映射

Hibernate类型	数据库类型	JDBC类型	类型注册
org.hibernate.type.StringType	字符串	VARCHAR	字符串,以
org.hibernate.type.MaterializedClob	字符串	CLOB	物化clob
org.hibernate.type.TextType	字符串	用LONGVARCHAR	文本
org.hibernate.type.CharacterType	char,java朗性格	char	char,java朗性格
org.hibernate.type.BooleanType	布尔	钻头	布尔,java朗布尔
org.hibernate.type.NumericBooleanType	布尔	整数,0是假的,1是正确的	数字逻辑
org.hibernate.type.YesNoType	布尔	字符,“N”/“N”是假,“Y”/“Y”是正确的。 大写的值写入到数据库。	是的没有
org.hibernate.type.TrueFalseType	布尔	CHAR,F / F是假的,'T'/'T'是真的。 大写的值写入到数据库。	真的假
org.hibernate.type.ByteType	字节,java朗字节	非常小的整数	字节,java朗字节
org.hibernate.type.ShortType	短,java朗短	SMALLINT	短,java朗短
org.hibernate.type.IntegerTypes	int,java . lang . integer	整数	int,java . lang . integer
org.hibernate.type.LongType	长,java朗长	长整型数字	长,java朗长
org.hibernate.type.FloatType	浮,java朗浮	FLOAT	float, java.lang.Float
org.hibernate.type.DoubleType	double, java.lang.Double	DOUBLE	double, java.lang.Double
org.hibernate.type.BigIntegerType	java.math.BigInteger	NUMERIC	big_integer
org.hibernate.type.BigDecimalType	java.math.BigDecimal	NUMERIC	big_decimal, java.math.bigDecimal
org.hibernate.type.TimestampType	java.sql.Timestamp	TIMESTAMP	timestamp, java.sql.Timestamp
org.hibernate.type.TimeType	java.sql.Time	TIME	time, java.sql.Time
org.hibernate.type.DateType	java.sql.Date	DATE	date, java.sql.Date
org.hibernate.type.CalendarType	java.util.Calendar	TIMESTAMP	calendar, java.util.Calendar
org.hibernate.type.CalendarDateType	java.util.Calendar	DATE	calendar_date
org.hibernate.type.CurrencyType	java.util.Currency	VARCHAR	currency, java.util.Currency
org.hibernate.type.LocaleType	java.util.Locale	VARCHAR	locale, java.utility.locale
org.hibernate.type.TimeZoneType	java.util.TimeZone	VARCHAR, using the TimeZone ID	timezone, java.util.TimeZone
org.hibernate.type.UrlType	java.net.URL	VARCHAR	url, java.net.URL
org.hibernate.type.ClassType	java.lang.Class	VARCHAR, using the class name	class, java.lang.Class
org.hibernate.type.BlobType	java.sql.Blob	BLOB	blog, java.sql.Blob
org.hibernate.type.ClobType	java.sql.Clob	CLOB	clob,java sql clob
org.hibernate.type.BinaryType	原始byte[]	VARBINARY	二进制,byte[]
org.hibernate.type.MaterializedBlobType	原始byte[]	BLOB	materized_blob
org.hibernate.type.ImageType	原始byte[]	LONGVARBINARY	形象
org.hibernate.type.BinaryType	java朗byte[]	VARBINARY	包装二进制
org.hibernate.type.CharArrayType	char[]	VARCHAR	字符,char[]
org.hibernate.type.CharacterArrayType	java朗字符[]	VARCHAR	包装器字符,字符[],java朗性格[]
org.hibernate.type.UUIDBinaryType	java util uuid	二进制	uuid二进制,java util uuid
org.hibernate.type.UUIDCharType	java util uuid	字符,也可以读VARCHAR	uuid char
org.hibernate.type.PostgresUUIDType	java util uuid	PostgreSQL UUID,通过类型#,这符合PostgreSQL JDBC驱动程序 定义	pg uuid
org.hibernate.type.SerializableType	实现者的java朗可串行化的	VARBINARY	不像其他的值类型,这种类型的多个实例注册。 这是注册 一旦在io。 可序列化的,注册在特定的io。 序列化实现 类名。

8 1 2。 复合类型

复合类型 ,或 嵌入的类型 ,因为他们被称为由Java 持久性API,传统上被称为 组件 在冬眠。 所有这些 术语的含义相同。

组件的值代表聚合到一个单独的Java类型。 一个例子是一个 地址 类,它聚合了街道、城市、国家和邮政编码。 复合类型 以类似的方式表现对一个实体。 他们都是专门为一个应用程序编写的类。 他们可能 既包括引用其他特定于应用程序的类,以及集合和简单的JDK 类型。 唯一的区分因素,一个组件不有自己的生命周期或定义一个 标识符。

8 1 3。 集合类型

一个 收集 式指的是数据类型本身,而不是它的内容。

一个集合代表一个一对一或一对多关系数据库的表。

参考章节收集更多的信息集合。

8.2。 实体类型

实体是特定于应用程序的类,它们相互关联表中的数据,使用一个惟一的标识符。 因为 的要求,一个独特的标识符,ntities独立存在和定义自己的生命周期。 作为一个 例,删除一个成员不应该删除用户或组。 有关更多信息,请参见章节 持久化类。

8.3。 不同的数据分类的影响

需要写

第9章。 映射实体

表的内容

9.1。层次结构

第十章。 映射关联

最基本形式的映射在Hibernate映射持久化实体类是一个数据库表。 你可以把它扩展概念映射相关的类在一起。 吗? ? ? 显示了一个 人 类与

第11章。 HQL和JPQL

表的内容

Hibernate查询语言(HQL)和Java持久性查询语言(JPQL)都是对象模型 集中查询语言本质上类似于SQL。 是一个严重受JPQL HQL的子集。 JPQL 查询一直是一个有效的HQL查询,相反的是不真实的然而。

两个HQL和JPQL都非类型安全的方式来执行查询操作。 标准查询提供了一个 类型安全的方法来查询。 看到 吗？？？ 为更多的信息。

11.1。 大小写敏感性

除了Java类和属性的名称,查询是不区分大小写的。 所以 选择 是一样的 选择 是一样的 选择 ,但 org hibernate如foo 和 org hibernate如foo 是不同的,是吗 foo.BARSET 和 foo.BARSET 。

> **注意**
>
> 本文档使用小写的关键词作为大会的例子。

11.2。 声明类型

两个HQL和JPQL允许 选择 , 更新 和 删除 语句被执行。 HQL另外允许 插入 语句,在一个形式 类似于SQL 插入选择 。

> **重要**
>
> 应注意,当一个 更新 或 删除 语句 执行。
>
> > 应谨慎使用在执行批量更新或删除操作,因为它们可能导致 之间不一致的数据库和实体在活跃的持久化上下文。 一般来说,散装 更新和删除操作只能在一个事务中执行新的持久性con - 文本或之前抓取或访问实体的状态会影响此类操作。
> >
> > —— **4.10节的JPA 2.0规范**

11 2 1。 Select语句

BNF的 选择 语句在HQL是:

```
select_statement :: =
    [select_clause]
    from_clause
    [where_clause]
    [groupby_clause]
    [having_clause]
    [orderby_clause]
```

最简单的可能的HQL 选择 语句的形式:

```
from com.acme.Cat
```

select语句在JPQL中的完全相同,除了需要一个HQL JPQL 选择条款 ,而HQL不。 尽管HQL不需要存在 的 选择条款 ,它通常包括一个良好的实践。 对于简单的查询 目的是明确的,所以想要的结果的 选择条款 是东 推断。 但在更复杂的查询,并非总是如此。 它通常是更好的明确 指定的意图。 Hibernate实际上并没有强迫 选择条款 在场 即使解析JPQL查询,但是应用程序感兴趣的JPA可移植性最好留心 这。

11 2 2。 Update语句

BNF的 更新 语句是相同的在HQL和JPQL:

```
update_statement ::= update_clause [where_clause]

update_clause ::= UPDATE entity_name [[AS] identification_variable]
```

```
        SET update_item {, update_item}*

update_item ::= [identification_variable.]{state_field | single_valued_object_field}
        = new_value

new_value ::= scalar_expression |
                simple_entity_expression |
                NULL
```

更新 语句,默认情况下,不影响 版本 或 时间戳 属性值为受影响的实体。 然而, 你可以迫使Hibernate来设置 版本 或 时间戳 属性 值通过使用 版本更新 。 这是通过添加 版本 关键字后 更新 关键字。 但是请注意, 这是一个特定的功能,不会冬眠工作在一个便携的方式。 定制版本类型, org.hibernate.usertype.UserVersionType ,在一起是不被允许的 与 更新版本 语句。

一个 更新 语句执行使用 executeupdate 要么 org hibernate查询 或 javax.persistence.Query 。 该方法被命名为那些熟悉 JDBC executeupdate 在 java sql preparedstatement 。 这个 int 返回的值 executeUpdate() 方法 表示数量的实体操作的影响。 这可能会或可能不会关联到号码 数据库中的影响的行。 一个HQL批量操作可能导致多个实际的SQL语句 被执行(加入子类,例如)。 返回的数字表示实际的数量 实体声明的影响。 使用一个连接的继承层次结构,对一个的删除 子类实际上可能导致删除对不仅仅是表的子类映射, 但也“根”表和表 “ 之间的 “

例11.1。 示例查询语句更新

```
String hqlUpdate =
                "update Customer c " +
                "set c.name = :newName " +
                "where c.name = :oldName";
int updatedEntities = session.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
```

```
String jpqlUpdate =
                "update Customer c " +
                "set c.name = :newName " +
                "where c.name = :oldName";
int updatedEntities = entityManager.createQuery( jpqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
```

```
String hqlVersionedUpdate =
                "update versioned Customer c " +
                "set c.name = :newName " +
                "where c.name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
```

> **重要**
>
> 无论是 更新 也 删除 语句可以 导致所谓的隐式连接。 他们已经不允许显式连接形式。

11 2 3。 删除语句

BNF的 删除 语句是相同的在HQL和JPQL:

```
delete_statement ::= delete_clause [where_clause]

delete_clause ::= DELETE FROM entity_name [[AS] identification_variable]
```

一个 删除 执行语句也使用 executeupdate 方法要么 org hibernate查询 或 javax.persistence.Query 。

11 2 4。 Insert语句

HQL添加能够定义 插入 语句也。 没有JPQL 相当于这个。 对于一个HQL的BNF 插入 声明:

```
insert_statement ::= insert_clause select_statement

insert_clause ::= INSERT INTO entity_name (attribute_list)

attribute_list ::= state_field[, state_field ]*
```

这个 属性列表 类似于 列规范 在 SQL 插入 语句。 对于实体参与绘制继承,只有属性 直接定义在命名实体的可用于属性列表 。 超类 属性不允许和子类的属性不会有意义。 换句话说, 插入 语句是固有的非多态。

select语句 可以是任何有效的HQL查询选择,但需要说明的是,返回吗 类型必须匹配类型所期望的插入。 目前,这是在查询检查编译而不是允许检查提交到数据库。 这可能导致问题 Hibernate类型之间 等效 而不是 平等 。 例如,这可能导致导致问题之间的不匹配 属性映射为一个 org.hibernate.type.DateType 和一个属性定义为 一个 org.hibernate.type.TimestampType , 即使数据库可能不让 区别或也许能够处理转换。

为id属性,insert语句给你两个选择。 您可以显式地指定 id属性 属性列表 ,在这种情况下,它的价值来自于 相应的选择表达,或

省略它 属性列表 在这种情况下,一个 使用生成的值。 这后一个选项只有当使用id发电机操作 “ 在数据库 ” ,试图使用这个选项与任何 “ 在内存中 “ 类型 发电机将导致异常在解析。

乐观锁的属性,insert语句再给你两个选择。 你可以 指定属性 属性列表 在这种情况下,它的值是来自 相应的选择表达式,或省略它 属性列表 在这 以防 种子值 定义为相应的 org.hibernate.type.VersionType 是使用。

例11.2。 示例查询语句插入

```
String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert ).executeUpdate();
```

11.3。 从 条款

这个 从 条款负责定义的对象范围模型类型可用 其余的查询。 它还负责定义所有的 “ 识别变量 ” 提供给其他查询。

11 3 1。 识别变量

识别变量通常被称为别名。 引用对象模型类 在FROM子句中可以关联到一个识别变量,然后可以使用 请参考该类型乔纳其余的查询。

在大多数情况下,声明一个识别变量是可选的,尽管它通常是良好的实践 声明它们。

一个识别变量必须遵守规则对于Java标识符有效性。

根据JPQL,识别变量必须被视为不区分大小写。 良好的实践 说你应该使用相同的情况在整个查询引用给定的识别变量。 在 句话说,JPQL说他们 可以 区分大小写,所以Hibernate必须 可以这样对待他们,但这并不能说明它很好实践。

11 3 2。 根实体引用

一根实体引用,或者JPA调用 范围变量声明 ,是 特别是引用映射实体类型的应用程序。 它不能名称组件/ 可嵌入的类型。 和协会,包括集合,以不同的方式处理 稍后讨论。

为根的BNF实体的引用:

```
root_entity_reference ::= entity_name [AS] identification_variable
```

例11.3。 简单的查询示例

```
select c from com.acme.Cat c
```

我们看到,查询是定义一个根实体的引用 com acme猫 对象模型类型。 此外,它声明的一个别名 C 来, com acme猫 参考;这是识别变量。

通常根实体引用只是名称的 实体名称 而不是 实体类FQN。 默认情况下,实体的名字是不合格的实体类名称, 这里 猫

例11.4。 简单的查询使用实体名称为根实体引用

```
select c from Cat c
```

多个根实体引用也可以被指定。 即使命名相同的实体!

例11.5。 简单的查询使用多个根实体引用

```
// build a product between customers and active mailing campaigns so we can spam!
select distinct cust, camp
from Customer cust, Campaign camp
where camp.type = 'mail'
  and current_timestamp() between camp.activeRange.start and camp.activeRange.end
```

```
// retrieve all customers with headquarters in the same state as Acme's headquarters
select distinct c1
from Customer c1, Customer c2
where c1.address.state = c2.address.state
  and c2.name = 'Acme'
```

11 3 3。 显式连接

这个 从 条款也可以包含显式关系连接使用 加入 关键字。 这些连接可以是 内在 或 左外 风格连接。

例11.6。 显式内加入的例子

```
select c
from Customer c
    join c.chiefExecutive ceo
where ceo.age < 25

// same query but specifying join type as 'inner' explicitly
select c
from Customer c
    inner join c.chiefExecutive ceo
where ceo.age < 25
```

例11.7。 显式左(外)加入的例子

```
// get customers who have orders worth more than $5000
// or who are in "preferred" status
select distinct c
from Customer c
    left join c.orders o
where o.value > 5000.00
  or c.status = 'preferred'

// functionally the same query but using the
// 'left outer' phrase
select distinct c
from Customer c
    left outer join c.orders o
where o.value > 5000.00
  or c.status = 'preferred'
```

一个重要的用例是定义显式连接 获取连接 这覆盖 这个懒惰的加入协会。 作为一个例子,考虑到一个实体命名 客户 有收藏价值的协会命名的 订单

例11.8。 获取连接的例子

```
select c
from Customer c
    left join fetch c.orders o
```

正如您可以看到的从这个例子中,一个取加入关键字指定注射 取 在关键字 加入 。 在这个例子中,我们使用一个左外连接,因为我们想要的 返回客户没有订单也。 内连接也可以被获取。 但内部连接仍然 过滤器。 在这个例子中,使用内连接相反会导致客户没有任何订单 被过滤的结果。

> **重要**
>
> 获取连接是无效的在子查询。
> 应该小心当取加入收藏价值协会是任何进一步 限制;所获取的集合将被限制了! 因为这个原因,它通常被认为是 最佳实践不分配一个标识变量来获取连接除了目的 指定嵌套获取连接。 获取连接中不应使用分页查询(又名, setFirstResult / setMaxResults)。 也不应使用HQL 的 滚动 或 迭代 特性。

HQL还定义了一个 与 条款,才有资格加入条件。 再一次,这是 具体到HQL;JPQL没有定义这个特性。

例11.9。 与条款加入例子

```
select distinct c
from Customer c
    left join c.orders o
        with o.value > 5000.00
```

最重要的区别是,在生成的SQL的条件 与条款 是部分的吗 在条款 在生成的SQL 相对于其他的查询在本节中,HQL / JPQL条件的一部分 where子句 在生成的SQL。 在这个具体的例子的区别是 可能不是那么重要。 这个 与条款 有时候是必须的吗 复杂的查询。

显式连接可能参考协会或组件/嵌入式属性。 为进一步的信息 关于收藏价值协会引用,看到 节11 3 5, “集合成员引用” 。 对于组件/嵌入式属性,连接只是逻辑,不关联到一个 物理(SQL)加入。

11 3 4。 隐式连接(路径表达式)

另一种方法增加的对象范围模型类型可供查询是通过 使用隐式连接,或路径表达式。

例11.10。 简单隐式连接的例子

```
select c
from Customer c
where c.chiefExecutive.age < 25

// same as
select c
from Customer c
    inner join c.chiefExecutive ceo
where ceo.age < 25
```

一个隐式连接始终就一个 识别变量 ,其次是 导航操作符(.),其次是一个属性的类型引用的对象模型 初始 识别变量 。 在这个例子中,最初的 识别变量 是 C 这指的 客户 实体。 这个 c的首席执行官 参考然后指 到 首席 属性的 客户 实体。 首席 是一个关联类型所以我们进一步导航到它吗 年龄 属性。

> **重要**
>
> 如果这个属性代表一个实体协会(非集合)或一个组件/嵌入式, 引用可以进一步导航。 基本价值和收藏价值的关联不能 进一步的导航。

的例子所示,隐式连接会出现外 FROM子句 。 然而, 他们影响 FROM子句 。 隐式连接总是视为内部连接。 多个引用相同的隐式连接总是指向相同的逻辑和物理(SQL)加入。

例11.11。 重用隐式连接

```
select c
from Customer c
where c.chiefExecutive.age < 25
   and c.chiefExecutive.address.state = 'TX'

// same as
select c
from Customer c
    inner join c.chiefExecutive ceo
where ceo.age < 25
  and ceo.address.state = 'TX'

// same as
select c
from Customer c
    inner join c.chiefExecutive ceo
    inner join ceo.address a
where ceo.age < 25
  and a.state = 'TX'
```

就像使用显式连接、隐式连接可能参考协会或组件/嵌入式属性。 为进一步的信息关于收藏价值协会引用,看到 节11 3 5, “集合成员引用” 。 对于组件/嵌入式属性, 连接是简单的逻辑,不关联到一个物理(SQL)加入。 与显式连接, 然而,隐式连接也可以参考基本状态字段只要路径表达式结束 那里。

11 3 5。 集合成员引用

收藏价值实际上引用关联参考 值 的 这个集合。

例11.12。 收集引用的例子

```
select c
from Customer c
    join c.orders o
    join o.lineItems l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'

// alternate syntax
select c
from Customer c,
    in(c.orders) o,
    in(o.lineItems) l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'
```

在这个例子中,识别变量 o 实际上指的对象模型 类型 秩序 这是元素的类型的吗 客户订单号 协会。

这个示例还演示备用语法来指定收藏协会加入使用 在 语法。 两种形式是等价的。 形成一个应用程序选择使用是吗 简单的味道。

11 3 5 1。 特殊情况——限定路径表达式

我们早先说过,收藏价值协会实际上参考 值 的集合。 基于集合的类型,也有可用的一组 明确的资格表达式。

例11.13。 合格收集引用的例子

```
// Product.images is a Map<String,String> : key = a name, value = file path

// select all the image file paths (the map value) for Product#123
select i
from Product p
    join p.images i
where p.id = 123

// same as above
select value(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names (the map key) for Product#123
select key(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names and file paths (the 'Map.Entry') for Product#123
select entry(i)
from Product p
    join p.images i
where p.id = 123

// total the value of the initial line items for all orders for a customer
select sum( li.amount )
from Customer c
       join c.orders o
       join o.lineItems li
where c.id = 123
  and index(li) = 1
```

价值

指的是收藏价值。 不指定限定符一样。 有用 明确地表明意图。 有效的对任何类型的收藏价值的参考。

指数

根据HQL规则,这是有效的对于地图和列表,指定一个 javax.persistence.OrderColumn 注释引用 地图键或列表的位置(即OrderColumn值)。 JPQL然而,储备 这个列表中使用案例和补充道 关键 在这个地图的情况下。 应用程序感兴趣的JPA提供者可移植性应该意识到这一点 的区别。

关键

有效的只有地图。 指地图的关键。 如果键本身就是一个实体, 可以进一步导航。

条目

只有有效的只有地图。 指地图的逻辑 java util映射条目 元组(组合的关键 和值)。 条目 只有有效的作为一个终端的路径,只有有效吗 在select子句。

看到 节11 4 9”,收集相关的表情” 收集相关的额外细节 表达式。

11 3 6。多态性

HQL和JPQL查询本身是多态的。

```
select p from Payment p
```

这个查询的名称 付款 实体的明确。 然而,所有的子类 付款 也可用来查询。 所以如果 CreditCardPayment 实体和WireTransferPayment 实体 每个扩展从 付款 所有三种类型可以查询。 和 该查询将返回实例的所有三个。

> **逻辑极端**
>
> HQL查询的 从java . lang . object 完全是有效的! 它返回每个 每一种类型的对象定义在你的应用程序。

这是可以改变的,要么通过 org hibernate注释多态性 注释(全球,和 hibernate特定)或限制他们使用在查询本身使用一个实体类型表达式。

11.4。表达式

基本上表达式引用,解决基本的或元组值。

11 4 1。 识别变量

看到 11.3节,“ 从 条款” 。

11 4 2。 路径表达式

再次,看到 11.3节,“ 从 条款” 。

11 4 3。文字

括在单引号字符串。 一个单引号来逃避在一个字符串,使用 双单引号。

例11.14。 字符串的例子

```
select c
from Customer c
where c.name = 'Acme'

select c
from Customer c
where c.name = 'Acme''s Pretzel Logic'
```

数字字面值是允许在几个不同的形式。

例11.15。 数值文字的例子

```
// simple integer literal
select o
from Order o
where o.referenceNumber = 123

// simple integer literal, typed as a long
select o
from Order o
where o.referenceNumber = 123L

// decimal notation
select o
from Order o
where o.total > 5000.00

// decimal notation, typed as a float
select o
from Order o
where o.total > 5000.00F

// scientific notation
select o
from Order o
where o.total > 5e+3

// scientific notation, typed as a float
select o
from Order o
where o.total > 5e+3F
```

在科学的符号形式, E 不区分大小写。

特定的输入可以通过使用的相同的后缀的方法指定的Java。 所以, l 代表一个长; D 代表一个双; f 代表一个浮动。 实际的后缀是不分大小写。

布尔文字是 真正的 和 假 ,又不区分大小写的。

枚举甚至可以引用为文字。 完全限定类名必须使用枚举。 HQL 还可以处理常量以同样的方式,虽然JPQL没有定义,作为支持。

实体名称也可以被用作文字。 看到 节11 4 10”,实体类型” 。

日期/时间文字可以指定使用JDBC逃脱的语法: { d ' yyyy mm dd ' } 对日期、 { t “hh:mm:ss ' } 对于时间和 { ts的hh:mm yyyy mm dd:ss(米尔斯)} (从而可选)为时间戳。 这些 文字只有工作如果你JDBC驱动程序支持他们。

11 4 4。参数

HQL支持所有3以下形式。 JPQL不支持hql具体位置 参数的概念。 是一个好习惯,不是混合在一个给定的查询形式。

11 4 4 1。 命名参数

命名参数来声明一个冒号,后面是一个标识符 :aNamedParameter 。 同样的命名参数可以在一个查询中出现多次。

例11.16。 命名参数的例子

```
String queryString =
        "select c " +
        "from Customer c " +
        "where c.name = :name " +
        "   or c.nickName = :name";

// HQL
List customers = session.createQuery( queryString )
        .setParameter( "name", theNameOfInterest )
        .list();

// JPQL
List<Customer> customers = entityManager.createQuery( queryString, Customer.class )
        .setParameter( "name", theNameOfInterest )
        .getResultList();
```

11 4 4 2。 位置(JPQL)参数

jpql风格位置参数来声明一个问号后跟一个序数- ？1，？2 。 ordinals从1开始的。 就像 命名参数、位置参数也可以在一个查询中出现多次。

例11.17。 位置(JPQL)参数的例子

```
String queryString =
        "select c " +
        "from Customer c " +
        "where c.name = ?1 " +
        "   or c.nickName = ?1";

// HQL - as you can see, handled just like named parameters
//       in terms of API
List customers = session.createQuery( queryString )
        .setParameter( "1", theNameOfInterest )
        .list();

// JPQL
List<Customer> customers = entityManager.createQuery( queryString, Customer.class )
        .setParameter( 1, theNameOfInterest )
        .getResultList();
```

11 4 4 3。 位置(HQL)参数

hql风格位置参数遵循JDBC位置参数的语法。 他们被宣布使用 吗？ 没有以下顺序。 没有办法联系两个这样的 位置参数为 “同一” 除了绑定相同的值,每个。

这种形式应考虑弃用,可能在不久的将来被移除。

11 4 5。算术

算术运算也代表有效的表达式。

例11.18。 数值运算的例子

```
select year( current_date() ) - year( c.dateOfBirth )
from Customer c

select c
from Customer c
where year( current_date() ) - year( c.dateOfBirth ) < 30

select o.customer, o.total + ( o.total * :salesTax )
from Order o
```

以下规则适用于算术运算的结果:

- 如果这两个操作数是双/双,结果是一个双;
- 另外,如果这两个操作数是浮/浮动,结果是一个浮动;
- 另外,如果不是操作数是BigDecimal,结果是BigDecimal;
- 另外,如果不是操作数是入BigInteger,结果是入BigInteger(除了分区 这种情况下结果类型不是进一步明确);
- 另外,如果不是操作是长/长,结果是长(除了部门,在 这种情况下结果类型不是进一步明确);
- 别的,(即假定两个操作数都是整数类型)结果是整数 (除了部门,在这种情况下,结果类型不是进一步明确);

日期算术也受到支持,但更有限的时尚。 这是由于部分 数据库支持的差异和部分缺乏支持 间隔 定义的查询语言本身。

11 4 6。 连接(操作)

HQL定义了一个连接运营商除了支持连接（ CONCAT)函数。 这不是由JPQL,所以可移植的应用程序 应避免使用。 连接操作是来自SQL连接操作符- | | 。

例11.19。 拼接操作示例

```
select 'Mr. ' || c.name.first || ' ' || c.name.last
from Customer c
where c.gender = Gender.MALE
```

看到 节11 4 8,“标量函数” 详细讨论 concat() 函数

11 4 7。 聚合函数

聚合函数也有效的表达式在HQL和JPQL。 语义是一样的 SQL同行。 支持的聚合函数是:

- 计数 (包括不同的/所有预选赛)——结果类型总是长。
- AVG ——结果类型始终是双重的。
- 分钟 ——结果类型是相同的类型作为参数。
- 马克斯 ——结果类型是相同的类型作为参数。
- 金额 ——结果的类型 avg() 函数取决于 值的类型平均。 对于积分值(除了入BigInteger),结果 类型是长。 对于浮点值(除了BigDecimal)结果类型是双。 对于 入BigInteger值,结果类型是入BigInteger。 对于BigDecimal值,结果类型BigDecimal。

例11.20。 聚合函数的例子

```
select count(*), sum( o.total ), avg( o.total ), min( o.total ), max( o.total )
from Order o

select count( distinct c.name )
from Customer c

select c.id, c.name, sum( o.total )
from Customer c
    left join c.orders o
group by c.id, c.name
```

聚合和分组经常出现。 关于分组看到 11.8节,“分组”

11 4 8。 标量函数

两个HQL和JPQL定义一些标准函数,可不管底层 数据库在使用。 HQL也能理解附加功能定义的方言以及 应用程序。

11 4 8 1。 标准化的功能——JPQL

下面是函数的列表定义为支持JPQL。 应用感兴趣的剩余 便携式JPA提供者之间应该坚持这些函数。

CONCAT

字符串连接函数。 变量参数长度的2个或多个字符串值 被连接在一起。

substring

提取字符串值的一部分。

```
substring( string_expression, numeric_expression [, numeric_expression] )
```

第二个参数表示起始位置。 第三个(可选参数) 表示长度。

上

上情况下指定的字符串

低

较低的情况下指定的字符串

修剪

遵循的语义SQL trim函数。

长度

返回一个字符串的长度。

定位

定位一个字符串在另一个字符串。

```
locate( string_expression, string_expression[, numeric_expression] )
```

第三个参数(可选)是用来表示一个位置开始寻找。

abs

计算数学绝对价值的一个数值。

国防部

计算的剩余部分将第一个参数的第二个。

sqrt

计算数学平方根的一个数值。

当前日期

返回数据库的当前日期。

当前时间

返回数据库当前时间。

当前时间戳的值

返回数据库当前时间戳。

11 4 8 2。 标准化的功能——HQL

超出了JPQL标准化的功能,使得一些额外的可用功能HQL不管 底层数据库的使用。

钻头长度

返回长度的二进制数据。

铸

执行一个SQL铸。 演员的目标应该命名的Hibernate映射类型使用。 看到这一章的数据类型的更多信息。

提取

执行一个SQL萃取在datetime值。 一个萃取精华的部分 日期时间(年,例如)。 看到下面的缩写形式。

第二

缩写形式提取,提取第二个。

分钟

缩写形式提取提取一分钟。

小时

缩写形式提取提取小时。

天

缩写形式提取提取天。

月

缩写形式提取提取月。

年

缩写形式提取提取一年。

str

缩写形式,铸造一个值为字符数据。

11 4 8 3。 非标准功能

Hibernate方言可以注册附加功能已知用于特定 数据库产品。 这些功能也可以在HQL(JPQL,虽然只有当使用 Hibernate作为JPA提供者明显)。 然而,他们也只能当使用, 数据库/方言。 应用程序对数据库的可移植性目标应该避免使用功能 在这个类别。

应用程序开发人员也可以提供自己的函数集。 这将通常代表 要么自定义SQL函数或别名片段的SQL。 这样的函数声明是 由使用 addSqlFunction 方法 org hibernate cfg配置

11 4 9。 收集相关表达式

有一些专门的表达式使用收藏价值关联。 一般 这些只是缩写形式或其它表达式为了简洁。

大小

计算一个集合的大小。 相当于一个子查询!

maxelement

可以使用集合的基本类型。 是指确定的最大价值 通过应用 马克斯 SQL聚合。

maxindex

可供使用的索引集合。 指最大指数(键/位置) 取决于应用 马克斯 SQL聚合。

minelement

可以使用集合的基本类型。 指的是最小值确定 通过应用 分钟 SQL聚合。

minindex

可供使用的索引集合。 是指最低指数(键/位置) 取决于应用 分钟 SQL聚合。

元素

指集合中的元素作为一个整体。 只允许在where子句中。 经常结合使用 所有，任何 或 一些 限制。

指数

类似 元素 除了 指数 指 集合指数(键/职位)作为一个整体。

例11.21。 收集相关表达式的例子

```
select cal
from Calendar cal
where maxelement(cal.holidays) > current_date()

select o
from Order o
where maxindex(o.items) > 100

select o
from Order o
where minelement(o.items) > 10000

select m
from Cat as m, Cat as kit
where kit in elements(m.kittens)

// the above query can be re-written in jpql standard way:
select m
from Cat as m, Cat as kit
where kit member of m.kittens

select p
from NameList l, Person p
where p.name = some elements(l.names)

select cat
from Cat cat
where exists elements(cat.kittens)

select p
from Player p
where 3 > all elements(p.scores)

select show
from Show show
where 'fizard' in indices(show.acts)
```

元素的索引集合(数组,列表和地图)可以被索引操作符。

例11.22。 索引操作符的例子

```
select o
from Order o
where o.items[0].id = 1234

select p
from Person p, Calendar c
where c.holidays['national day'] = p.birthDay
  and p.nationality.calendar = c

select i
from Item i, Order o
where o.items[ o.deliveredItemIndices[0] ] = i
  and o.id = 11

select i
from Item i, Order o
where o.items[ maxindex(o.items) ] = i
  and o.id = 11

select i
```

```
from Item i, Order o
where o.items[ size(o.items) - 1 ] = i
```

看到也 节11 3 5 1, “特例-限定路径表达式” 因为有大量的重叠。

11 4 10。 实体类型

我们也可以参考类型的实体作为一个表达式。 这主要是有用的在处理 用实体继承层次结构。 表达的类型可以使用 类型 函数 指一个识别变量的类型代表一个实体。 的名字 实体也可以作为一个方法来引用一个实体类型。 另外实体类型可以 参数化,在这种情况下,实体的Java类引用将被绑定的参数 值。

例11.23。 实体类型表达式的例子

```
select p
from Payment p
where type(p) = CreditCardPayment

select p
from Payment p
where type(p) = :aType
```

HQL也有一个遗留形式的指一个实体类型,尽管这遗产形式是考虑 不赞成赞成 类型 。 遗留的形式将使用 p类 的例子中,而不是 类型(p) 。 它是只提到的完整性。

11 4 11。 案例表达式

两个简单的和搜索表单的支持,以及2 SQL定义缩写形式 (NULLIF 和 合并)

11 4 11 1。 简单情况表情

简单的表格下面的语法:

```
CASE {operand} WHEN {test_value} THEN {match_result} ELSE {miss_result} END
```

例11.24。 简单case表达式的例子

```
select case c.nickName when null then '<no nick name>' else c.nickName end
from Customer c

// This NULL checking is such a common case that most dbs
// define an abbreviated CASE form.  For example:
select nvl( c.nickName, '<no nick name>' )
from Customer c

// or:
select isnull( c.nickName, '<no nick name>' )
from Customer c

// the standard coalesce abbreviated form can be used
// to achieve the same result:
select coalesce( c.nickName, '<no nick name>' )
from Customer c
```

11 4 11 2。 搜索案例表达式

搜索形式有以下的语法:

```
CASE [ WHEN {test_conditional} THEN {match_result} ]* ELSE {miss_result} END
```

例11.25。 搜查case表达式的例子

```
select case when c.name.first is not null then c.name.first
            when c.nickName is not null then c.nickName
            else '<no first name>' end
from Customer c

// Again, the abbreviated form coalesce can handle this a
// little more succinctly
```

```
select coalesce( c.name.first, c.nickName, '<no first name>' )
from Customer c
```

11 4 11 3。 NULLIF表达式

是一个缩写NULLIF CASE表达式,返回NULL如果其操作数,被认为是相等的。

例11.26。 NULLIF例子

```
// return customers who have changed their last name
select nullif( c.previousName.last, c.name.last )
from Customer c

// equivalent CASE expression
select case when c.previousName.last = c.name.last then null
            else c.previousName.last end
from Customer c
```

11 4 11 4。 合并表达式

合并是一个缩写CASE表达式,返回第一个非空操作数。 我们已经看到一个 结合上述的例子的数量。

11.5。 选择 条款

这个 选择 条款确定哪些对象和值作为查询结果返回。 这个表达式讨论 11.4节,“表情” 都是有效的选择表情,除了吗 另有声明。 看到部分 11.10节,“查询API” 信息处理结果 根据不同的类型的值中指定的 选择 条款。

有一个特定的表达式类型,只有有效的select子句中。 Hibernate调用这个 “ 动态实例化 ” 。 JPQL支持一些功能的和调用它 一个 “ 构造函数表达式 ”

例11.27。 动态实例化的例子——构造函数

```
select new Family( mother, mate, offspr )
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

因此,而不是处理对象[](再一次,看到 11.10节,“查询API”)我们到了包装 中的值类型安全的java对象,将返回查询的结果。 类引用必须完全合格的,它必须有一个匹配的构造函数。

这里的课程不需要映射。 如果它确实代表一个实体,生成的实例 返回的新状态(不成功!)。

这是部分支持也JPQL。 HQL支持额外的 “ 动态实例化 ” 特性。 首先,查询可以指定返回一个列表,而不是一个对象[]为标量结果:

例11.28。 动态实例化的例子——列表

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

从这个查询的结果将是一个列表<列表>与< Object[]>列表

HQL还支持包装的标量结果在地图上。

例11.29。 动态实例化的例子——地图

```
select new map( mother as mother, offspr as offspr, mate as mate )
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr

select new map( max(c.bodyWeight) as max, min(c.bodyWeight) as min, count(*) as n )
from Cat c
```

从这个查询的结果将是一个列表< Map < String,对象> >而不是一个 列表< Object[]>。 的键映射定义的别名给选择 表达式。

11.6。谓词

谓词形式基础的where子句,having子句和搜索案例表达式。 它们的表达式,决心一个真值,一般 真正的 或 假 ,尽管布尔比较涉及取消通常决心 未知 。

11 6 1。 关系比较

比较涉及的比较运算符- =、>、> =、<、< =、< >]>。 HQL还定义了 [CDATA[< ！！ =作为一个比较运算符< >的代名词。 操作数应该 相同的类型。

例11.30。 关系比较的例子

```
// numeric comparison
select c
from Customer c
where c.chiefExecutive.age < 30

// string comparison
select c
from Customer c
where c.name = 'Acme'

// datetime comparison
select c
from Customer c
where c.inceptionDate < {d '2000-01-01'}

// enum comparison
select c
from Customer c
where c.chiefExecutive.gender = com.acme.Gender.MALE

// boolean comparison
select c
from Customer c
where c.sendEmail = true

// entity type comparison
select p
from Payment p
where type(p) = WireTransferPayment

// entity value comparison
select c
from Customer c
where c.chiefExecutive = c.chiefTechnologist
```

比较也可以包括子查询限定符- 所有，任何，一些 。 一些和任何是同义的。

全限定符解析为真如果比较适用于所有的值的结果 子查询。 它解析为假如果子查询结果为空。

例11.31。 所有的子查询比较限定符的例子

```
// select all players that scored at least 3 points
// in every game.
select p
from Player p
where 3 > all (
    select spg.points
    from StatsPerGame spg
    where spg.player = p
)
```

任何/一些限定符解析为真如果比较适用于一些(至少一个) 值的子查询的结果。 它解析为假如果子查询结果为空。

11 6 2。 Nullness谓词

为nullness检查一个值。 可以应用于基本属性引用、实体引用和 参数。 HQL另外允许它被应用于组件/可嵌入的类型。

例11.32。 Nullness检查示例

```
// select everyone with an associated address
select p
```

```
from Person p
where p.address is not null

// select everyone without an associated address
select p
from Person p
where p.address is null
```

11 6 3。 像谓词

执行一个像比较字符串值。 语法是:

```
like_expression ::=
        string_expression
        [NOT] LIKE pattern_value
        [ESCAPE escape_character]
```

遵循的语义表达的SQL像。 这个 模式价值 是 模式匹配的尝试 字符串表达式 。 就像SQL, 模式价值 可以使用 “ _ ” 和 “ % ” 作为通配符。 这个 含义是相同的。 “ _ ” 匹配任何单个的字符。 “ % ” 匹配 任意数量的字符。

可选的 转义字符 用于指定一个转义字符用于 逃避的特殊意义 “ _ ” 和 “ % ” 在 模式价值 。 这是有用,当需要搜索模式包括要么 “ _ ” 或 “ % ”

例11.33。 像谓词的例子

```
select p
from Person p
where p.name like '%Schmidt'

select p
from Person p
where p.name not like 'Jingleheimmer%'

// find any with name starting with "sp_"
select sp
from StoredProcedureMetadata sp
where sp.name like 'sp|_%' escape '|'
```

11 6 4。 谓词之间

类似于SQL表达式之间。 执行一个评价,一个值范围内 2其他值。 所有的操作数应该有类似的类型。

例11.34。 谓词之间的例子

```
select p
from Customer c
    join c.paymentHistory p
where c.id = 123
  and index(p) between 0 and 9

select c
from Customer c
where c.president.dateOfBirth
        between {d '1945-01-01'}
            and {d '1965-01-01'}

select o
from Order o
where o.total between 500 and 5000

select p
from Person p
where p.name between 'A' and 'E'
```

11 6 5。 在谓词

在 谓词执行一个检查一个特定的值是在一列值。 它的语法是:

```
in_expression ::= single_valued_expression
            [NOT] IN single_valued_list

single_valued_list ::= constructor_expression |
            (subquery) |
            collection_valued_input_parameter

constructor_expression ::= (expression[, expression]*)
```

的类型 单值的表达式 和个人价值 单值的列表 必须是一致的。 JPQL限制了有效的类型在这里 对字符串、数字、日期、时间、时间戳和枚举类型。 在JPQL, 单值的表达式 只能参考:

- “ 状态字段 “,这是它的术语,简单的属性。 这具体 不包括协会和组件/嵌入式属性。
- 实体类型表达式。 看到 节11 4 10”,实体类型”

在HQL, 单值的表达式 可以引用一个更广泛的一系列表达式 类型。 单值允许协会。 所以是组件/嵌入式属性,尽管这 功能取决于级别的支持tuple或 “ 行值的构造函数的语法 “ 在 底层数据库。 此外,HQL并未限制值类型以任何方式,虽然 应用程序开发人员应该意识到不同类型可能招致有限支持基于 底层的数据库供应商。 这主要的原因限制。JPQL

列表的值可以来自许多不同的来源。 在 构造函数表达式 和 收藏价值的输入参数 , 列值不能是空的,它必须包含至少一个值。

例11.35。 在谓词的例子

```
select p
from Payment p
where type(p) in (CreditCardPayment, WireTransferPayment)

select c
from Customer c
where c.hqAddress.state in ('TX', 'OK', 'LA', 'NM')

select c
from Customer c
where c.hqAddress.state in ?

select c
from Customer c
where c.hqAddress.state in (
    select dm.state
    from DeliveryMetadata dm
    where dm.salesTax is not null
)

// Not JPQL compliant!
select c
from Customer c
where c.name in (
    ('John','Doe'),
    ('Jane','Doe')
)

// Not JPQL compliant!
select c
from Customer c
where c.chiefExecutive in (
    select p
    from Person p
    where ...
)
```

11 6 6。 存在谓词

存在表达式测试结果的存在从子查询。 肯定的形式返回true 如果子查询结果包含值。 否定的形式返回true,如果子查询 结果是空的。

11 6 7。 空集合谓词

这个 是[不]空吗 表达式适用于收集价值路径表达式。 它 检查是否有任何关联的特定集合值。

例11.36。 空集合表达式的例子

```
select o
from Order o
where o.lineItems is empty

select c
from Customer c
where c.pastDueBills is not empty
```

11 6 8。 谓词集合的成员

这个 [不]成员[] 表达式适用于收集价值路径表达式。 它 检查是否一个值是一个指定的集合的成员。

例11.37。 成员的集合表达式的例子

```
select p
from Person p
where 'John' member of p.nickNames

select p
from Person p
where p.name.first = 'Joseph'
  and 'Joey' not member of p.nickNames
```

11 6 9。 没有谓语运营商

这个 不 操作符用于否定谓语,遵循它。 如果这 以下是真实的谓词,没有解析为false。 如果谓词是真的,不立志 假。 如果谓词是未知的,不解决也未知。

11 6 10。 和谓词运营商

这个 和 操作符用于将2谓词表达式。 结果 和表达是真的当且仅当两个谓词解析为真的。 如果任一谓词解析 未知,和表达式解析为未知也。 否则,结果是假的。

11 6 11。 或谓词运营商

这个 或 操作符用于将2谓词表达式。 结果 或表达式是正确的如果不是谓词解析为真。 如果两个谓词解决未知, 或表达式解析为未知。 否则,结果是假的。

11.7。 在 条款

这个 在 条款的一个查询的谓词,其断言是否值 每个潜在的行匹配断言检查。 因此,where子句限制返回的结果 从一个select查询和限制范围的更新和删除查询。

11.8。分组

这个 集团 条款允许构建聚合结果对各种价值的组织。 作为一个 例子,可以考虑以下几点:

例11.38。 集团通过插图

```
// retrieve the total for all orders
select sum( o.total )
from Order o

// retrieve the total of all orders
// *grouped by* customer
select c.id, sum( o.total )
from Order o
    inner join o.customer c
group by c.id
```

第一个查询检索完成所有订单的总额。 第二次检索总对于每个 客户,按每个客户。

在一个分组查询,where子句应用于非聚合值(从本质上说,它决定 行进入聚合)。 这个 有 条款也限制了结果, 但是它是在聚合值。 在 例11.38, “集团通过插图” 的例子, 我们检索订单总数为所有客户。 如果最终被太多的数据要处理, 我们可能想要限制结果只专注客户提供一个总结订单总额超过 10000元:

例11.39。 有插图

```
select c.id, sum( o.total )
from Order o
    inner join o.customer c
group by c.id
having sum( o.total ) > 10000.00
```

HAVING子句遵循相同的规则作为WHERE子句,也是由谓词。 有是 应用后的分组和聚合已经完成,在应用之前。

11.9。订购

查询的结果还可以命令。 这个 订单 子句用于规定 选中的值被用来命令结果。 表达式的类型被认为是有效的部分 类型的排序的条款包括:

- 状态字段
- 组件/可嵌入属性
- 标量表达式,如算术运算、函数等等。
- 识别变量声明在select子句为前面的任何表达式类型

此外,JPQL说,所有值类型的排序中引用条款必须命名的选择 条款。 HQL不强制要求限制,但应用程序数据库的可移植性应该渴望 意识到并不是所有的数据库都支持引用值类型的排序条款,没有引用 在select子句。

单个表达式的类型的排序可以合格,要么 asc (提升)或 Desc (降序)来表示所需的排序方向。

例11.40。 类型的排序的例子

```
// legal because p.name is implicitly part of p
select p
from Person p
order by p.name

select c.id, sum( o.total ) as t
from Order o
    inner join o.customer c
group by c.id
order by t
```

11.10。 查询API

第十二章。标准

表的内容

标准查询提供类型安全的替代HQL,JPQL和原生sql查询。

> **重要**
>
> Hibernate提供了一个年长的、遗留 org hibernate标准 API应该 考虑弃用。 没有功能开发将目标这些api。 最终,hibernate特定 标准特性将被移植到JPA扩展 javax.persistence.criteria.CriteriaQuery 。 详细讨论 org hibernate标准 API,看到 吗? ? ? 。
> 本章将集中在JPA api为声明类型安全的标准查询。

标准查询是一个程序性、类型安全的方式来表达查询。 他们是类型安全的方面 使用接口和类来表示各种结构部分查询如查询本身, 或select子句,或者一个类型的排序,等等。他们也可以在引用属性类型安全条款 正如我们将会看到在一个位。 用户老一辈的冬眠 org hibernate标准 查询API将识别的一般方法,虽然我们相信JPA API是优越的 因为它代表了一个干净的看的经验教训,API。

标准查询本质上是一个对象图,每个部分的图代表一个增加 (当我们浏览下这个图)更多的原子部分查询。 第一步在执行标准查询 这个图是建筑。 这个 javax.persistence.criteria.CriteriaBuilder 接口是第一件事,你需要知晓开始使用标准查询。 它的角色是一个工厂的所有各个部分的标准。 你获得一个 javax.persistence.criteria.CriteriaBuilder 实例通过调用 getCriteriaBuilder 方法要么 javax.persistence.EntityManagerFactory 或 javax.persistence.EntityManager 。

下一步是获得一个 javax.persistence.criteria.CriteriaQuery 。 这 是通过使用3种方法之一 javax.persistence.criteria.CriteriaBuilder 为了这个目的:

```
<T> CriteriaQuery<T> createQuery(Class<T> resultClass);
CriteriaQuery<Tuple> createTupleQuery();
CriteriaQuery<Object> createQuery();
```

每个服务于不同的目的根据预期类型的查询结果。

> **注意**

第六章标准API JPA规范的 已经包含相当多的参考材料用于修饰或说明的不同部分 标准查询。 因此,而不是复制所有内容在这里,让相反看一些的 更广泛预期用途的API。

12.1。 类型化标准查询

条件查询的类型(又名< T >)显示预期的类型的查询 结果。 这可能是一个实体,一个整数,或任何其他对象。

12个1 1。 选择一个实体

这可能是最常见的查询。 应用程序想要选择实体实例。

例12.1。 选择根实体

```
CriteriaQuery<Person> criteria = builder.createQuery( Person.class );
Root<Person> personRoot = criteria.from( Person.class );
criteria.select( personRoot );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Person> people = em.createQuery( criteria ).getResultList();
for ( Person person : people ) {
    ...
}
```

本例使用 createQuery 传递 人 类引用作为查询的结果将Person对象。

> **注意**
>
> 在调用 criteriaquery选择 方法在这个例子 不必要的,因为 personRoot 将隐含的选择因为我们吗 只有一个查询根。 这是在这里完成只有完整性的一个例子。
> 这个 Person_.eyeColor 引用一个例子的静态形式的JPA 元模型参考。 我们将使用该表格只在这一章。 看到 文档的Hibernate JPA Metamodel发生器的额外细节 JPA静态元模型。

12个1 2。 选择一个表达式

最简单的形式的选择一个表达式是选择一个特定的属性从一个实体。 但这种表达也可能代表一个聚合、数学运算等。

例12.2。 选择一个属性

```
CriteriaQuery<Integer> criteria = builder.createQuery( Integer.class );
Root<Person> personRoot = criteria.from( Person.class );
criteria.select( personRoot.get( Person_.age ) );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Integer> ages = em.createQuery( criteria ).getResultList();
for ( Integer age : ages ) {
    ...
}
```

在本例中,查询类型的 java . lang . integer 因为这 是预期的结果(类型的类型 人#年龄 属性 是 java . lang . integer)。 因为一个查询可能包含多个引用 人的实体,属性引用总是需要合格。 这是完成的 根#得到 方法调用。

12个1 3。 选择多个值

实际上有一些不同的方法来选择多个值使用标准查询。 我们 将探索2选项,但另一种推荐的方法是使用元组中描述的吗 12.2节,“元组标准查询” 。 或者考虑一个包装器查询;请参阅 节12 1 4”,选择一个包装器” 详情。

例12.3。 选择数组

```
CriteriaQuery<Object[]> criteria = builder.createQuery( Object[].class );
Root<Person> personRoot = criteria.from( Person.class );
Path<Long> idPath = personRoot.get( Person_.id );
Path<Integer> agePath = personRoot.get( Person_.age );
criteria.select( builder.array( idPath, agePath ) );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Object[]> valueArray = em.createQuery( criteria ).getResultList();
```

```
for ( Object[] values : valueArray ) {
        final Long id = (Long) values[0];
        final Integer age = (Integer) values[1];
        ...
}
```

从技术上讲这是归类为一个类型的查询,但是你可以看到,从处理结果 这是一种误导。 无论如何,预期的结果类型是数组。

然后使用的例子 数组 方法 javax.persistence.criteria.CriteriaBuilder 曾明确 结合个人的选择到一个 javax.persistence.criteria.CompoundSelection 。

例12.4。 选择一个数组(2)

```
CriteriaQuery<Object[]> criteria = builder.createQuery( Object[].class );
Root<Person> personRoot = criteria.from( Person.class );
Path<Long> idPath = personRoot.get( Person_.id );
Path<Integer> agePath = personRoot.get( Person_.age );
criteria.multiselect( idPath, agePath );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Object[]> valueArray = em.createQuery( criteria ).getResultList();
for ( Object[] values : valueArray ) {
    final Long id = (Long) values[0];
    final Integer age = (Integer) values[1];
    ...
}
```

正如我们看到的 例12.3,“选择一个数组” 我们有一个类型的标准 查询返回一个对象数组。 这两个查询都在功能上是等同的。 这第二个例子 使用 多选 方法考察基础上略有不同 当标准类型给定查询始建,但是在本例中它说选择和 返回一个 Object[] 。

12个1 4。 选择包装

另一个替代 节12 1 3,“选择多个值” 是相反 选择一个对象,该对象将 “ 包装 “ 的多个值。 回到这个例子 查询,而不是返回一个数组 [人# id、人#年龄) 相反声明一个类,它拥有这些值,而回报。

例12.5。 选择一个包装器

```
public class PersonWrapper {
        private final Long id;
        private final Integer age;
        public PersonWrapper(Long id, Integer age) {
                this.id = id;
                this.age = age;
        }
        ...
}

...

CriteriaQuery<PersonWrapper> criteria = builder.createQuery( PersonWrapper.class );
Root<Person> personRoot = criteria.from( Person.class );
criteria.select(
                builder.construct(
                        PersonWrapper.class,
                        personRoot.get( Person_.id ),
                        personRoot.get( Person_.age )
                )
);
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<PersonWrapper> people = em.createQuery( criteria ).getResultList();
for ( PersonWrapper person : people ) {
    ...
}
```

首先我们看到的简单定义包装器对象我们将使用包装我们的结果 值。 特别注意到构造函数及其参数类型。 因为我们将返回 PersonWrapper 对象,我们使用 PersonWrapper 随着 类型的我们的标准查询。

这个例子演示了使用 javax.persistence.criteria.CriteriaBuilder 方法 构造 这是用于创建一个包装器表达式。 每一行都在 因此我们说我们想一个 PersonWrapper 实例 其余参数的匹配的构造函数。 这个包装器表达式然后通过 这个选择。

12.2。 元组标准查询

一个更好的方法来 节12 1 3,“选择多个值” 是使用要么 包装(我们看到的 节12 1 4”,选择一个包装器”)或使用 javax.persistence.Tuple 合同。

例12.6。 选择一个元组

```
CriteriaQuery<Tuple> criteria = builder.createTupleQuery();
Root<Person> personRoot = criteria.from( Person.class );
Path<Long> idPath = personRoot.get( Person_.id );
Path<Integer> agePath = personRoot.get( Person_.age );
criteria.multiselect( idPath, agePath );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Tuple> tuples = em.createQuery( criteria ).getResultList();
for ( Tuple tuple : valueArray ) {
    assert tuple.get( 0 ) == tuple.get( idPath );
        assert tuple.get( 1 ) == tuple.get( agePath );
        ...
}
```

这个例子说明了访问查询结果通过 javax.persistence.Tuple 接口。 这个例子使用显式的 createTupleQuery 的 javax.persistence.criteria.CriteriaBuilder 。 另一种方法 是使用 createQuery 传递 tuple类 。

我们又一次看到使用 多选 方法,就像在 例12.4,“选择一个数组(2)” 。 这里的差异是,类型的 javax.persistence.criteria.CriteriaQuery 被定义为 javax.persistence.Tuple 所以在本例中是复合的选择 解释是元组元素。

这个 javax.persistence.Tuple 合同提供3形式的访问 底层的元素:

类型

这个 例12.6,“选择一个元组” 示例说明了这种形式的访问 在 元组。 get(idPath) 和 元组。 get(agePath) 调用。 这允许类型化访问底层元组值的基础上 javax.persistence.TupleElement 表达式用来构建 这个标准。

位置

允许访问底层元组值基于位置。 简单的 对象得到(int位置) 形式非常类似于访问 示 例12.3,“选择一个数组” 和 例12.4,“选择一个数组(2)” 。 这个 < X > X get(int位置、类< X >类型 形式 允许输入位置访问,但基于显式地提供类型的元组 值必须是类型分配到。

别名

允许访问底层基于元组值(可选)指定别名。 这个 示例查询并不适用于一个别名。 别名是应用通过 别名 方法 javax.persistence.criteria.Selection 。 就像 位置 访问中,都有一个类型 (对象get(String别名))和非 (< X > X get(String别名、类< X >类型 形式。

12.3。 FROM子句

> 一个对象定义了一个查询CriteriaQuery在一个或多个实体,可嵌入或基本抽象 模式类型。 根对象是实体的查询,其他类型是达到了 通过导航。
>
> —— **JPA规范** ,部分6 5 2查询根、pg 262

> **注意**
>
> 所有的个人部分从条款(根、连接、路径)实现 javax.persistence.criteria.From 接口。

12 3 1。根

根定义基础上,所有的连接、路径和属性查询中是可用的。 根始终是一个实体类型。 根是定义并添加到标准的重载 从 方法 javax.persistence.criteria.CriteriaQuery :

```
<X> Root<X> from(Class<X>);

<X> Root<X> from(EntityType<X>)
```

例12.7。 添加根

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
// create and add the root
person.from( Person.class );
```

标准查询可以定义多个根,其结果是创建一个笛卡儿 产品在新添加的根和其他人。 下面是一个示例匹配所有单 男人和所有单身女性:

例12.8。 添加多个根

```
CriteriaQuery query = builder.createQuery();
Root<Person> men = query.from( Person.class );
Root<Person> women = query.from( Person.class );
Predicate menRestriction = builder.and(
                builder.equal( men.get( Person_.gender ), Gender.MALE ),
                builder.equal( men.get( Person_.relationshipStatus ), RelationshipStatus.SINGLE )
```

```
);
Predicate womenRestriction = builder.and(
        builder.equal( women.get( Person_.gender ), Gender.FEMALE ),
        builder.equal( women.get( Person_.relationshipStatus ), RelationshipStatus.SINGLE )
);
query.where( builder.and( menRestriction, womenRestriction ) );
```

12 3 2。连接

加入允许导航从其他 javax.persistence.criteria.From 要么协会或嵌入属性。 连接是由众多的重载 加入 方法 javax.persistence.criteria.From 接口

例12.9。 与嵌入式和ManyToOne例子

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
// Person.address is an embedded attribute
Join<Person,Address> personAddress = personRoot.join( Person_.address );
// Address.country is a ManyToOne
Join<Address,Country> addressCountry = personAddress.join( Address_.country );
```

例12.10。 示例集合

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
Join<Person,Order> orders = personRoot.join( Person_.orders );
Join<Order,LineItem> orderLines = orders.join( Order_.lineItems );
```

12 3 3。取

就像在HQL和JPQL,标准查询可以指定相关数据被获取沿着 与业主。 获取是由众多的重载 取 方法 javax.persistence.criteria.From 接口。

例12.11。 与嵌入式和ManyToOne例子

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
// Person.address is an embedded attribute
Fetch<Person,Address> personAddress = personRoot.fetch( Person_.address );
// Address.country is a ManyToOne
Fetch<Address,Country> addressCountry = personAddress.fetch( Address_.country );
```

> **注意**
>
> 技术上来说,嵌入式属性总是拿来和他们的主人。 然而在 为了定义抓取的 地址#国家 我们需要一个 javax.persistence.criteria.Fetch 对于它的父路径。

例12.12。 示例集合

```
CriteriaQuery&lt;Person&gt; personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
Fetch<Person,Order> orders = personRoot.fetch( Person_.orders );
Fetch<Order,LineItem> orderLines = orders.fetch( Order_.lineItems );
```

12.4。 路径表达式

> **注意**
>
> 根,连接和读取路径本身。

12.5。 使用参数

例12.13。 使用参数

```
CriteriaQuery<Person> criteria = build.createQuery( Person.class );
Root<Person> personRoot = criteria.from( Person.class );
criteria.select( personRoot );
ParameterExpression<String> eyeColorParam = builder.parameter( String.class );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), eyeColorParam ) );

TypedQuery<Person> query = em.createQuery( criteria );
query.setParameter( eyeColorParam, "brown" );
List<Person> people = query.getResultList();
```

使用 参数 方法 javax.persistence.criteria.CriteriaBuilder 为了获得一个参数 参考。 然后使用参数引用绑定参数值 javax.persistence.Query

第13章。 原生SQL查询

表的内容

你也可以表达查询本机SQL方言你的数据库。 这是有用的,如果你 想利用数据库特定的功能,如查询提示或连接的选项在Oracle。 它还提供了一个干净的迁移路径从一个直接基于SQL / JDBC应用程序Hibernate / JPA。 Hibernate还允许您指定手写的SQL(包括存储过程)所有 创建、更新、删除和加载操作。

13.1。 使用 SqlQuery

执行本地SQL查询是通过控制 SqlQuery 接口,它是通过调用 Session.createSQLQuery() 。 以下部分 描述如何使用这个API查询。

13 1 1。 标量查询

最基本的SQL查询来获得一个列表的标量 (值)。

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

这将返回一个列表的对象数组(Object[])与标量 值表中每列的猫。 Hibernate将使用 推导出实际的订单ResultSetMetadata和类型的返回 标量值。

避免使用的开销 ResultSetMetaData ,或者只是更明确的在 返回的是什么,一个人可以用吗 addScalar() :

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

这个查询指定:

- SQL查询字符串
- 列和类型返回

这将返回对象数组,但现在不会使用 ResultSetMetaData 反而会显式地获得 ID、姓名和出生年月日分别列一长,字符串和一个短的 从底层的结果集。 这也意味着只有这三个 列将被归还,即使查询使用 * 和可能的回报要高于3家上市 列。

它可以保留全部或部分类型信息 标量的。

```
sess.createSQLQuery("SELECT * FROM CATS")
```

```
    .addScalar("ID", Hibernate.LONG)
    .addScalar("NAME")
    .addScalar("BIRTHDATE")
```

这本质上是相同的查询和之前一样,但现在 ResultSetMetaData 是用于确定类型的吗 的名字和出生年月日,随着类型的ID是明确的 指定的。

如何在java . sql。 类型映射回来ResultSetMetaData 对Hibernate类型控制的方言。 如果一个特定的类型 不映射,或不会导致预期的类型,它是可能的 通过调用定制它 registerHibernateType 在 方言。

13 1 2。 实体查询

上面的查询都是关于返回标量值, 基本上返回的“原始”值从resultset。 以下 显示了如何获取实体对象从原生sql查询通过addEntity() 。

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

这个查询指定:

- SQL查询字符串
- 查询返回的实体

假设猫被映射为一个类中的列ID、名称 和出生年月日上面的查询都将返回一个列表,其中每个 元素是一个猫的实体。

如果实体映射与 多对一 到 另一个实体必须还回这当执行 原生查询,否则数据库特定的“列未找到”的错误 将会发生。 额外的列将自动时返回 使用*符号,但我们宁愿被显式的,如下面 作为例子 多对一 一个 狗 :

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

这将允许cat.getDog()功能的正常运行。

13.1.3节。 处理协会和集合

它是可能的加入。急切地 狗 到 避免可能的额外往返初始化代理。 这是 通过 addJoin() 方法,它允许你 加入一个协会或集合。

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d WHE
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

在本例中,返回的 猫 ' s将 他们 狗 属性完全初始化没有任何 额外的往返到数据库。 请注意,您添加了一个别名 (“猫”)能够指定目标属性的路径加入。 它 可以做同样的渴望加入为集合,例如如果 猫 有一对多, 狗 相反。

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

在这个阶段你到达无限可能与 本地查询,没有开始提高sql查询来让他们 可用在冬眠。 问题可能出现在返回多个实体 同一类型的或默认的别名/列名不 足够的。

13 1 4。 返回多个实体

直到现在,结果集列名都假设是相同的 指定的列名称的映射文档。 这可以 有问题的SQL查询,连接多个表,因为相同的 列名可以出现在多个表。

列别名注射需要以下查询(很可能会失败):

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

查询的目的是返回两个猫每行实例:一只猫 和它的母亲。 查询将,然而,失败,因为有一个 名称冲突;实例映射到相同的列的名称。 同时,在一些数据库返回的列别名最有可能 表单上的“c。 ID”、“c。 名称”等等,这不是等于列 指定的映射 (“ID”和“名”)。

下面的表格是不容易列名 重复:

```
sess.createSQLQuery("SELECT {cat.*}, {m.*}  FROM CATS c, CATS m WHERE c.MOTHER_ID = m.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

这个查询指定:

- SQL查询字符串,使用占位符为Hibernate来 注入列别名
- 查询返回的实体

{猫。 * }和{母亲。 * }符号上面使用的是一个简称 “所有的属性”。 或者,你可以列出明确的列, 但即使在这种情况下Hibernate注入SQL列别名为每个 财产。 一个列别名的占位符是属性名 合格的表别名。 在接下来的例子中,您检索 猫和他们的母亲从一个不同的表(猫日志)到一个 宣布在映射元数据中。 你甚至可以使用属性别名 在where子句中。

```
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()
```

13 1 4 1。 别名和属性引用

在大多数情况下上面的别名注射是必要的。 查询 涉及更复杂的映射,如复合属性, 继承鉴别器,收藏等,您可以使用特定的 别名,让Hibernate注入适当的别名。

下表显示了不同的方法可以使用 别名注射。 请注意,这些别名的结果 简单的例子,每个别名将有一个独特的和可能不同 当使用的名字。

表13.1。 别名注射的名字

描述	语法	例子
一个简单的属性	{[aliasname]。[返回一个只读字符串)	一个名字为{项目名称}
复合属性	{[aliasname]。[componentname]。[返回一个只读字符串]}	货币作为{项目金额。 货币},值 {项目金额值}
鉴别器的一个实体	{[aliasname]. class }	盘为{条目类}
实体的所有属性	{[aliasname]。* }	{项目。* }
一个收集关键	{[aliasname]键}	ORGID为{胶原关键}
一个集合的id	{[aliasname]id }	EMPID为{胶原id }
元素的集合	{[aliasname]元素}	XID为{胶原元素}
属性的元素的集合	{[aliasname]元素。[返回一个只读字符串]}	名作为{胶原元素名称}
所有属性的元素的集合	{[aliasname]元素。* }	{胶原元素。* }
所有属性的集合	{[aliasname]。* }	{科尔。* }

13 1 5。 返回非受管实体

它是可能的应用ResultTransformer原生SQL查询, 让它返回非受管实体。

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

这个查询指定:

- SQL查询字符串
- 由于变压器

上面的查询将返回一个列表的 CatDTO 已经实例化和注入值名称和 BIRTHNAME到相应属性或字段。

13 1 6。 处理继承

本地的SQL查询,查询实体映射作为 遗产的一部分必须包括所有的属性和baseclass 所有的子类。

13 1 7。参数

原生SQL查询支持位置以及命名 参数:

```
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();
```

13.2。 命名的SQL查询

SQL查询的命名也可以被定义在映射文件和 叫以完全相同的方式作为一个名叫HQL查询(参见 吗？？？)。 在这种情况下,你做的 不 需要调用 addEntity() 。

例13.1。 命名为sql查询使用< sql查询>映射 元素

```
<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>
```

例13.2。 执行一个已命名查询

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

这个 <返回加入> 元素是用于连接 协会和 <负载收集> 元素是 用于定义查询的初始化集合,

例13.3。 与协会命名的sql查询

```
<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>
```

一个名叫SQL查询可能返回标量值。 你必须申报 列别名和Hibernate类型使用 <返回标量> 元素:

例13.4。 命名查询返回一个标量

```
<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>
```

你可以外部化resultset映射信息 < resultset > 元素将允许你 要么重用它们在几个命名查询或通过 setResultSetMapping() API。

例13.5。 < resultset >映射用于外部化映射 信息

```
<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>
```

你可以,另外,使用结果集映射信息 你的hbm文件直接在java代码。

例13.6。 以编程方式指定结果的映射信息

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

到目前为止我们只有看着外化SQL查询使用 Hibernate映射文件。 同样的概念也可提供 anntations和被称为命名本机查询。 您可以使用 @NamedNativeQuery (@NamedNativeQueries)结合 @SqlResultSetMapping (@SqlResultSetMappings)。 像 @NamedQuery , @NamedNativeQuery 和 @SqlResultSetMapping 可以定义在类的级别, 但是他们是应用程序的全局范围。 让我们看看一个视图 的例子。

例13.7,“命名的SQL查询使用 @NamedNativeQuery 连同 @SqlResultSetMapping ”展示了如何 resultSetMapping 参数定义在 @NamedNativeQuery 。 它代表了一个定义的名称 @SqlResultSetMapping 。 resultset映射声明 实体检索到这原生查询。 每个字段的实体 绑定到一个SQL别名(或列名称)。 所有领域的实体包括 的子类和外键列相关的实体 必须出现在SQL查询。 字段定义是可选的 只要他们映射到相同的列名称作为一个上声明 类属性。 在示例2的实体, 晚上 和 区域 ,返回和每个属性声明和 关联到一个列名称,实际上列名称检索的 查询。

在 例13.8,“隐式结果集映射” 结果 集映射是隐式的。 我们只描述实体类的结果 集映射。 属性/列映射是通过使用实体 映射值。 在这种情况下,模型属性绑定到模型三种 列。

最后,如果协会一个相关的实体包括复合 主键, @FieldResult 元素应该用于 每一个外键列。 这个 @FieldResult 叫 由属性名的关系,其次是一个点（“。”),其次是名称或域或属性的主键。 中可以看到这个 例13.9,“使用点符号在@FieldResult指定关联 ”。

例13.7。 命名为SQL查询使用 @NamedNativeQuery 连同 @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")
    })
    }
)
```

例13.8。 隐式结果集映射

```
@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;
    }
}
```

例13.9。 在@FieldResult使用点符号用于指定关联

```
@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 surfac
    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;
    }
}
```

提示

如果你检索一个单一的实体使用默认的映射,您可以 指定 resultClass 属性而不是 resultSetMapping :

```
@NamedNativeQuery(name="implicitSample", query="select * from SpaceShip", resultCl
public class SpaceShip {
```

在你的一些本地查询,你将不得不返回标量值, 例如在构建报告查询。 你可以在地图 @SqlResultSetMapping 通过 @ColumnResult 。 你真的可以混合,实体和 标量返回相同的原生查询(这可能是不常见的 虽然)。

例13.10。 标量值通过 @ColumnResult

```
@SqlResultSetMapping(name="scalar", columns=@ColumnResult(name="dimension"))
@NamedNativeQuery(name="scalar", query="select length*width as dimension from SpaceShip", resultSetM
```

另一个特定于本地查询查询提示介绍: org hibernate可调用的 这可能或真或假的 取决于查询是一个存储过程或不是。

13 2 1。 使用返回属性来显式地指定列/别名 名称

你可以明确告诉Hibernate使用什么列别名 <返回属性> ,而不是使用 { } 语法让Hibernate注入自己的别名 示例:

```
<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>
```

<返回属性> 也适用于 多个列。 这解决了一个限制的 { } 语法不能允许细粒度的控制 多列属性。

```
<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>
```

在这个例子 <返回属性> 是 用于结合 { } 语法对于注射。 这允许用户选择他们想怎样参考列和 属性。

如果你的映射有鉴别器必须使用 <返回鉴频器> 指定 鉴频器列。

13 2 2。 使用存储过程查询

支持查询Hibernate3通过存储过程和 功能。 最下面的文档都是等价的。 存储过程/函数必须返回一个记录集作为第一 出参数能够使用Hibernate。 这样的一个例子 存储函数在Oracle 9和更高的如下:

```
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;
```

使用这个查询在Hibernate你需要地图通过命名 查询。

```
<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>
```

存储过程只返回标量和实体目前。 <返回加入> 和 <负载收集> 不支持。

13 2 2 1。 规则/限制使用存储过程

你不能使用存储过程与Hibernate除非你 过程/函数遵循一些规则。 如果他们不遵守这些 他们无法使用规则与Hibernate。 如果你仍然想要使用 这些程序你必须执行它们通过 会话连接() 。 规则是不同的 每个数据库,从数据库厂商有不同的存储过程语义/句法。

存储过程查询不能分页与 setFirstResult()/ setMaxResults() 。

推荐的调用形式是SQL92标准: { ? =叫 functionName(<参数>)} 或 { ? =叫 procedureName(<参数> } 。 本地调用语法不是 支持。

对于Oracle以下规则适用:

- 一个函数必须返回一个结果集。第一个参数 一个程序必须是一个 出 返回一个 结果集。这是通过使用一个 sys refcursor 类型在Oracle 9或10。 在Oracle 您需要定义一个 REF光标 类型。 看到 甲骨文文学为进一步的信息。

对于Sybase或MS SQL server以下规则适用:

- 这个过程必须返回一个结果集。注意,因为 这些服务器可以返回多个结果集和更新计数, Hibernate将迭代结果和采取的第一个结果 是一个结果集作为它的返回值。 一切将会 丢弃。
- 如果你能使 NOCOUNT设置在 在你的 程序,它可能会更有效率,但这不是一个 要求。

13.3。 自定义的SQL创建、更新和删除

可以使用自定义的SQL Hibernate3为创建、更新和删除 操作。 SQL可以覆盖在语句级或 如果列水平。 本节描述语句覆盖。 对于 列,看到 吗? ? ? 。 例13.11, “定制CRUD通过注释” 显示了如何定义 自定义的SQL operatons使用注释。

例13.11。 定制CRUD通过注释

```
@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 CH
public class Chaos {
    @Id
    private Long id;
    private Long size;
    private String name;
    private String nickname;
```

@SQLInsert , @SQLUpdate , @SQLDelete , @SQLDeleteAll 分别覆盖插入、更新、删除、全部删除 语句。 同样可以通过使用Hibernate映射文件和 < sql insert > , < sql更新> 和 < sql删除> 节点。 中可以看到这个 例13.12, “定制CRUD XML” 。

例13.12。 定制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>
```

如果您希望调用一个存储过程,一定要设置 可调用的 属性 真正的 。 在 注释和xml。

检查执行情况正确,Hibernate允许你 定义的三个策略:

- 没有:没有执行检查:存储过程预计将 失败在问题
- 数:使用rowcount检查更新 成功的

- 参数:如计数但使用输出参数相当, 标准机制

定义结果检查风格,使用 检查 参数是同样可以在annoations以及xml。

您可以使用相同的组注释xml节点分别 收集相关语句覆盖的曲线图 例13.13,“重写SQL语句的集合使用 注释” 。

例13.13。 重写SQL语句的集合使用 注释

```
@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>();
```

> **提示**
>
> 参数的顺序非常重要和被定义为订单 Hibernate处理属性。 你可以看到预期的订单通过启用 调试日志记录 org hibernate持续程序实体 水平。 这个水平使Hibernate将打印出静态SQL 这是用于创建、更新、删除等实体。 (看 预期的序列,记得不包括您的自定义SQL通过注释或映射文件,将覆盖Hibernate 生成的静态sql)

重写SQL语句对于二次表也是可能的 使用 @org.hibernate.annotations.Table ,要么(或 所有的)属性 sqlInsert , SqlUpdate , sqlDelete :

例13.14。 重写SQL语句对于二次表

```
@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.SELE
        sqlInsert=@SQLInsert(sql="insert into Cat2(storyPart2, id) values(upper(?), ?)") )
} )
public class Cat implements Serializable {
```

前面的示例还显示,你可以给一个评论到 给定表(原发性或继发性):这个注释将用于DDL 一代。

> **提示**
>
> 直接执行的SQL数据库中,所以您可以使用任何 方言你喜欢。 不过,这会降低你的可移植性如果你使用数据库特定的映射SQL。

最后但并非最不重要,存储过程是在大多数情况下要求 返回的行数插入、更新和删除。 Hibernate总是 注册第一个语句参数作为一个数字输出参数 反刍的操作:

例13.15。 存储过程和他们的返回值

```
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;
```

13.4。 定制SQL加载

你也可以声明自己的SQL(或HQL)查询实体 加载。 对于插入、更新和删除操作,这可以做的 个人专栏中描述的水平 吗? ? ? 或在语句级。 这里 是一个例子,一个语句级别覆盖:

```
<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>
```

这只是一个命名查询声明,正如前面所讨论的。 你 可以参考这个命名查询在一个类映射:

```
    <class name="Person">
        <id name="id">
            <generator class="increment"/>
        </id>
        <property name="name" not-null="true"/>
        <loader query-ref="person"/>
    </class>
```

这甚至适用于存储过程。

你甚至可以定义一个查询收集加载:

```
    <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>
```

您还可以定义一个实体加载器加载一个集合加入 获取:

```
    <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>
```

注释等效 <装载机> 是 @Loader注释见 例13.11,“定制CRUD通过注释” 。

第十四章。JMX

Envers第十五章。

文摘

Hibernate的目的是提供历史版本控制Envers应用程序的实体数据。 多 像源控制管理工具,如Subversion或Git,Hibernate Envers管理的概念修正 如果你的应用程序数据通过使用审计表。 每个事务涉及到一个全球修订编号 可以用来识别组织的变化(就像一个变更集在源代码控制)。 作为修正 是全球性的,有一个修订号,你可以查询各种实体在那修改,检索一个吗 (部分)视图的数据库,修改。 你可以找到一个修订号有一个约会,和其他 相反,你可以得到一个修订日期的承诺。

表的内容

15日12 3。 审计表分区的例子

15.13。 Envers链接

15.1。基础知识

审核变更执行的一个实体,您只需要两件事情: hibernate-envers 罐类路径上和一个 @Audited 注释 在实体上。

> **重要**
>
> 不像在以前的版本中,您不再需要指定侦听器在Hibernate配置 文件。 只是把Envers jar的类路径是足够的,听众将注册 自动。

和这一切-您可以创建、修改和删除实体一如既往。 如果你看看生成的 模式为实体,或在数据持久化了冬眠,你会注意到,没有变化。 然而,对于每个审计实体,一个新表介绍- 实体表aud，该对象存储历史数据,当你提交一个事务。 Envers自动创建审计 表如果 auto 选项设置为 创建 , 创建滴 或 更新 。 否则,输出完整的数据库模式 以编程的方式,使用org.hibernate.tool.EnversSchemaGenerator 。 适当的DDL 语句也可以生成的Ant任务后来描述与本手册。

而不是注释整个类和审核所有的属性,你可以标注 只有一些持久性属性与 @Audited 。 这将导致只有 这些属性被审计。

审计(历史)的一个实体可以访问使用 AuditReader 接口,它 可以得到在一个开放的吗 EntityManager 或 会话 通过 这个AuditReaderFactory 。 看到这些类的javadoc详细讨论 提供功能。

15.2。配置

可以配置Hibernate的各个方面Envers行为,例如表名称,等等。

表15.1。 Envers配置属性

属性名	默认值	描述
org.hibernate.envers.audit_table_prefix		字符串,将前缀的名称,约
org.hibernate.envers.audit_table_suffix	aud	字符串,将追加到一个审 如果你审计实体与一个 来存储 历史数据。
org.hibernate.envers.revision_field_name	启	名称字段的审计实体,将
org.hibernate.envers.revision_type_field_name	REVTYPE	名称字段的审计实体,将 尔)。
org.hibernate.envers.revision_on_collection_change	真正的	应该生成一个修改当一 一对多的关系,或现场使
org.hibernate.envers.do_not_audit_optimistic_locking_field	真正的	当真正的,属性用于乐观 不会 存储;它通常没有意
org.hibernate.envers.store_data_at_delete	假	应该实体数据被存储在 性为空)。 这是通常不 而,它是更容易和更有效 存储两次)。
org.hibernate.envers.default_schema	空(同一模式如表被审计单位)	默认的模式名称,应该用 @AuditTable(模式= 单位。
org.hibernate.envers.default_catalog	空(同一目录如表被审计单位)	默认的目录名称,应该用 @AuditTable(catalog 正常。
org.hibernate.envers.audit_strategy	org.hibernate.envers.strategy.DefaultAuditStrategy	审计策略,应该坚持审计 一个替代 org.hibern 开始修订和最终修订。 ValidityAuditStrategy
org.hibernate.envers.audit_strategy_validity_end_rev_field_name	REVEND	列名称,将举行最后修订 有效性,如果使用。
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp	假	应该结束的时间戳修订 身。 这是有用的能够 需要列 这是在桌上。 用。
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name	REVEND_TSTMP	的时间戳列名称的最终 ValidityAuditStrategy org.hibernate.envers 值为true
org.hibernate.envers.use_revision_entity_with_native_id	真正的	布尔标志,确定战略修订 机。 如果当前的数 false。 org.hibernate.id.enha 1. org.hibernate 2. org.hibernate
org.hibernate.envers.track_entities_changed_in_revision	假	该实体类型,这期间修改 REVCHANGES 表,用于 订标识符 (外国关键 R 15 5 1, “跟踪实体名称 的修订” 。

org.hibernate.envers.global_with_modified_flag	假,可以单独与重载 @Audited(withModifiedFlag = true)	房地产应该修改标志被有属性一个额外的布尔在给定的修订。 当设置用 @Audited 注释。 性级” 和 部分15 7 3,
org.hibernate.envers.modified_flag_suffix	国防部	后缀为列存储”修改标得到改良的旗帜,列名

> **重要**
>
> 以下配置选项已经被添加最近和应当被视为实验:
>
> org.hibernate.envers.track_entities_changed_in_revision
>
> org.hibernate.envers.using_modified_flag
>
> org.hibernate.envers.modified_flag_suffix

15.3。 额外的映射注释

审计表的名称可以设置在每个实体的基础上,使用 @AuditTable 注释。 它可能是乏味的添加这 注释每一个审计实体,所以如果可能的话,最好使用一个前缀/后缀。

如果你有一个映射与二次表,审计表他们中将生成 同样的方式(通过添加前缀和后缀)。 如果您希望覆盖此行为, 您可以使用 @SecondaryAuditTable 和 @SecondaryAuditTables 注释。

如果你想覆盖审计行为的一些字段/属性继承 @MappedSuperclass 或在嵌入式组件,您可以 应用 @AuditOverride(s) 注释或使用网站的亚型 的组件。

如果你想审计关系映射与 onetomany + @JoinColumn ，请见 15.11节,“映射例外” 额外的描述 @AuditJoinTable 注释,你可能想要使用。

如果你想审计关系,目标实体不是审计(是这样,例如 连实体,不改变,不需要被审计),只是注释它 @Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED) 。 然后,当阅读历史 版本的实体的关系,始终指向“当前”相关的实体。

如果你想审计属性的一个超类的一个实体,它是没有明确审计(没有 @Audited 注释在任何属性或类),你可以列出 超类的 auditParents 属性的 @Audited 注释。 请注意, auditParents 功能已经被弃用。 使用 @AuditOverride(SomeEntity forClass =。 类,isAudited = true / false) 相反。

15.4。 选择审计策略

在基本配置它是重要的选择审计策略,将用于持久化 和检索审计信息。 有一个平衡的持久化和性能 性能的查询审计信息。 目前有两个审计策略。

1. 默认的审计策略持续审计数据连同一个开始复习。 为每一行 插入、更新或者删除按照审计表,一个或多个行被插入到审计 表,连同开始调整它的有效性。 行审计表从来没有 更新后的插入。 使用子查询的查询审计信息来选择适用 审计表中的行。 这些子查询是出了名的缓慢和困难指数。
2. 另一个选择是一个有效的审计策略。 这个策略存储开始修订和 最终修订审计信息。 对于每一行插入、更新或者删除按照审计表, 一个或多个行被插入到审计表,连同开始修订的 效度。 但同时,场的结束修订前审计行(如果可用) 将这个修订。 查询审计信息可以使用之间的开始和结束 修订”而不是子查询所使用的默认的审计策略。

   这一战略的结果是,坚持审计信息会有点慢, 因为涉及的额外的更新,但检索审计信息会快很多。 这可以通过添加额外的索引。

15.5。 修订日志

测井数据进行修正

当Envers开始一个新版本,它会创建一个新的 修订实体 哪个商店 修改的信息。 默认情况下,这包括只是

1. 修订号 ——一个积分值(int /整数 或 长/长)。 本质上的主键修改
2. 修改时间戳 ——无论是一个 长/长 或 java util日期 值代表即时在修。 当使用一个 java util日期 ,而不是一个长/长 对于 修改时间戳,照顾不来存储这一列的数据类型,将松散的精度。

Envers处理这个信息作为一个实体。 默认情况下它使用它自己的内部类来充当 实体,映射到 REVINFO 表。 然而,你可以提供你自己的方法去收集这些信息,这可能是非常有用的 捕捉额外的细节,例如谁做了一个改变或ip地址的请求来。 有 是两件事你需要做这个工作。

1. 首先,你需要告诉Envers关于您希望使用的实体。 你的实体必须使用 @org.hibernate.envers.RevisionEntity 注释。 它必须 上述定义2属性标注 @org.hibernate.envers.RevisionNumber 和 @org.hibernate.envers.RevisionTimestamp ,分别。 您可以扩展 从 org.hibernate.envers.DefaultRevisionEntity ,如果你愿意,继承所有 这些要求的行为。

   只需添加自定义修改实体作为你正常的实体。 Envers将“发现”。 注意 这是一个错误,有多个实体标记为 @org.hibernate.envers.RevisionEntity

2. 第二,你需要告诉Envers如何创建的实例修改的单位处理 的 newRevision 方法

org.jboss.envers.RevisionListener 接口。

你告诉Envers定制 org.hibernate.envers.RevisionListener 实现使用通过指定它 @org.hibernate.envers.RevisionEntity 注释,使用 价值 属性。 如果你 RevisionListener 类是无法从 @RevisionEntity (如存在于一个不同的 模块),设置 org.hibernate.envers.revision_listener 财产的完全 限定名。 定义的类名由配置参数修改实体的覆盖 价值 属性。

```
@Entity
@RevisionEntity( MyCustomRevisionListener.class )
public class MyCustomRevisionEntity {
    ...
}

public class MyCustomRevisionListener implements RevisionListener {
    public void newRevision(Object revisionEntity) {
        ( (MyCustomRevisionEntity) revisionEntity )...;
    }
}
```

另一种方法来使用 org.hibernate.envers.RevisionListener 是相反调用吗 getCurrentRevision 方法 org.hibernate.envers.AuditReader 接口获取当前的修订, 并填补它与所需的信息。 该方法接受一个 持续 参数指示 是否应该坚持修改实体从这个方法返回之前。 真正的 确保返回的实体访问它的标识符值(修订号),但修订 公司将坚持无论有任何审计实体改变。 假 意味着修订号将 空 ,但修改实体将持续 只有一些审计实体已经改变了。

例15.1。 存储用户名和修改的例子

ExampleRevEntity.java

```
package org.hibernate.envers.example;

import org.hibernate.envers.RevisionEntity;
import org.hibernate.envers.DefaultRevisionEntity;

import javax.persistence.Entity;

@Entity
@RevisionEntity(ExampleListener.class)
public class ExampleRevEntity extends DefaultRevisionEntity {
    private String username;

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }
}
```

ExampleListener.java

```
package org.hibernate.envers.example;

import org.hibernate.envers.RevisionListener;
import org.jboss.seam.security.Identity;
import org.jboss.seam.Component;

public class ExampleListener implements RevisionListener {
    public void newRevision(Object revisionEntity) {
        ExampleRevEntity exampleRevEntity = (ExampleRevEntity) revisionEntity;
        Identity identity =
            (Identity) Component.getInstance("org.jboss.seam.security.identity");

        exampleRevEntity.setUsername(identity.getUsername());
    }
}
```

15 5 1。 跟踪实体名称期间修改修改

默认情况下实体类型,已经改变了在每个修订不被跟踪。 这意味着 需要查询所有表存储审计数据为了检索更改在 指定的修订。 Envers提供了一种简单的机制,创建了 REVCHANGES 表用来存储实体名称的修改持久化对象。 单记录封装了修订 标识符(外键 REVINFO 表)和一个字符串值。

跟踪实体名称的修改可以通过三种不同的方式:

1. 集 org.hibernate.envers.track_entities_changed_in_revision 参数 真正的 。 在这种情况下 org.hibernate.envers.DefaultTrackingModifiedEntitiesRevisionEntity 将 被隐式地用作修订日志实体。
2. 创建一个自定义修改实体扩展 org.hibernate.envers.DefaultTrackingModifiedEntitiesRevisionEntity 类。

   ```
   @Entity
   @RevisionEntity
   public class ExtendedRevisionEntity
               extends DefaultTrackingModifiedEntitiesRevisionEntity {
       ...
   }
   ```

3. 马克一个适当的字段的自定义实体与修订 @org.hibernate.envers.ModifiedEntityNames 注释。 房地产是 需要的集<字符串> 类型。

   ```
   @Entity
   @RevisionEntity
   public class AnnotatedTrackingRevisionEntity {
       ...
   ```

```
    @ElementCollection
    @JoinTable(name = "REVCHANGES", joinColumns = @JoinColumn(name = "REV"))
    @Column(name = "ENTITYNAME")
    @ModifiedEntityNames
    private Set<String> modifiedEntityNames;

    ...
}
```

用户,选择一种方法,可以检索上面列出的所有实体修改的 利用API指定修改描述 部分15 7 4”,查询实体修改给定的修订” 。

用户也可以实现自定义跟踪修改实体类型的机制。 在这种情况下,他们 应当通过自己的实现 org.hibernate.envers.EntityTrackingRevisionListener 接口作为值 的 @org.hibernate.envers.RevisionEntity 注释。 EntityTrackingRevisionListener 接口公开了一个方法,该方法通知 每当审计实体实例被添加、修改或删除当前修订边界内。

例15.2。 自定义实现跟踪实体类中修改修改

CustomEntityTrackingRevisionListener.java

```
public class CustomEntityTrackingRevisionListener
        implements EntityTrackingRevisionListener {
    @Override
    public void entityChanged(Class entityClass, String entityName,
                              Serializable entityId, RevisionType revisionType,
                              Object revisionEntity) {
        String type = entityClass.getName();
        ((CustomTrackingRevisionEntity)revisionEntity).addModifiedEntityType(type);
    }

    @Override
    public void newRevision(Object revisionEntity) {
    }
}
```

CustomTrackingRevisionEntity.java

```
@Entity
@RevisionEntity(CustomEntityTrackingRevisionListener.class)
public class CustomTrackingRevisionEntity {
    @Id
    @GeneratedValue
    @RevisionNumber
    private int customId;

    @RevisionTimestamp
    private long customTimestamp;

    @OneToMany(mappedBy="revision", cascade={CascadeType.PERSIST, CascadeType.REMOVE})
    private Set<ModifiedEntityTypeEntity> modifiedEntityTypes =
                                        new HashSet<ModifiedEntityTypeEntity>();

    public void addModifiedEntityType(String entityClassName) {
        modifiedEntityTypes.add(new ModifiedEntityTypeEntity(this, entityClassName));
    }

    ...
}
```

ModifiedEntityTypeEntity.java

```
@Entity
public class ModifiedEntityTypeEntity {
    @Id
    @GeneratedValue
    private Integer id;

    @ManyToOne
    private CustomTrackingRevisionEntity revision;

    private String entityClassName;

    ...
}
```

```
CustomTrackingRevisionEntity revEntity =
    getAuditReader().findRevision(CustomTrackingRevisionEntity.class, revisionNumber);
Set<ModifiedEntityTypeEntity> modifiedEntityTypes = revEntity.getModifiedEntityTypes()
```

15.6。 跟踪实体属性级的变化

默认情况下,只有通过Envers存储信息的实体是版本修改。 这种方法允许用户创建审计查询基于历史的数据实体的属性。 有时是有用的额外元数据存储为每个修订,当你有兴趣还在 这个类型的变化,不仅对产生的值。 功能描述 部分15 5 1,“跟踪实体名称修改在修正” 使得它可以告诉哪些实体被修改在一定的修正。 这里描述特征则更进一步。 “修改标志”Envers启用跟踪哪些 属性被修改的审计实体在一个给定的修订。

跟踪实体属性级的变化可以通过:

1. 设置 org.hibernate.envers.global_with_modified_flag 配置 财产 真正的 。 全球开关会导致添加修改标志 对于所有审计属性在所有审计实体。

2. 使用 @Audited(withModifiedFlag = true) 在一个属性或一个实体。

这个功能的取舍的到来是一种增加的大小 审计表和一个非常小,几乎可以忽略不计,性能下降 在审计写道。 这是由于这样的事实,即每一个跟踪 属性必须有一篇布尔列 模式存储信息的修改的属性。 的 当然是Envers”工作来填补这些列,因此没有额外的工作 开发人员是必需的。 由于成本所提到的,这是推荐的 有选择地启用这个特性,需要时用的 上面描述的精细配置手段。

看到“修改标志”可以被利用,检查非常 简单的查询API,使用它们: 部分15 7 3,“查询修改,修改给定属性的实体” 。

15.7。查询

你能想到的历史数据有两个维度。 第一,水平- 是数据库的状态在一个给定的修订。 因此,您可以 查询实体作为他们在修订联合国第二——垂直-是 修订,在哪些实体改变。 因此,您可以查询修改, 在一个给定的实体改变。

查询在Envers类似于Hibernate标准查询,所以如果你是常见的, 使用Envers查询将会容易得多。

的主要限制当前查询的实现是您不能 遍历的关系。 你只能指定约束的id 相关的实体,只有在“拥有”方面的关系。 这不过 将改变在将来的版本中。

请注意,查询在审计数据将在很多情况下要慢得多 比相应的查询“活”的数据,因为它们涉及subselects相关。

在未来,查询可提高无论是在速度和可能性,当使用有效时间 审计策略,这是当存储都开始和结束的修改实体。 看到 吗？？？。

15 7 1。 查询实体类在一个给定的修订

的入口点这种类型的查询:

```
AuditQuery query = getAuditReader()
    .createQuery()
    .forEntitiesAtRevision(MyEntity.class, revisionNumber);
```

然后您可以指定约束条件,这需要靠实体返回,通过 增加的限制,从而可以得到使用 AuditEntity 工厂类。 例如,只选择实体,其中“name”属性 等于“约翰”:

```
query.add(AuditEntity.property("name").eq("John"));
```

和只选择实体,都与一个给定的实体:

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
// or
query.add(AuditEntity.relatedId("address").eq(relatedEntityId));
```

你可以限制结果的数量,命令他们,并设置聚合和预测 (除分组)以通常的方式。 当你查询完成后,您可以通过调用获得结果 getSingleResult() 或 getResultList() 方法。

一个完整的查询,可以找例子如下:

```
List personsAtAddress = getAuditReader().createQuery()
    .forEntitiesAtRevision(Person.class, 12)
    .addOrder(AuditEntity.property("surname").desc())
    .add(AuditEntity.relatedId("address").eq(addressId))
    .setFirstResult(4)
    .setMaxResults(2)
    .getResultList();
```

15 7 2。 查询修改,哪些实体给定类的改变

的入口点这种类型的查询:

```
AuditQuery query = getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true);
```

您可以添加约束该查询以同样的方式作为对前一个问题。 有一些额外的可能性:

1. 使用 AuditEntity.revisionNumber() 你可以指定约束条件,预测 和订单的修订号,经审计的实体被修改
2. 同样,使用 AuditEntity.revisionProperty(返回一个只读字符串) 你可以指定约束, 预测和订单上的一个属性修改实体, 对应修改 的审计实体修改
3. AuditEntity.revisionType() 给你访问如上的类型的 修改(添加、国防部、德尔)。

通过使用这些方法, 你可以订单查询结果通过修订号,设置投影或约束 修订号码是大于或小于指定值,等等。例如, 以下查询将选择最小的修订号,实体类 MyEntity 与id entityId 已经改变了,在复习吗 42号:

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.id().eq(entityId))
    .add(AuditEntity.revisionNumber().gt(42))
    .getSingleResult();
```

第二个附加功能可以在查询中使用修改的能力 到maximalize /最小化属性。 例如,如果你想选择 修订,该值的 actualDate 对于一个给定的实体 大于给定值,但尽可能小:

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    // We are only interested in the first revision
```

```
        .setProjection(AuditEntity.revisionNumber().min())
        .add(AuditEntity.property("actualDate").minimize()
            .add(AuditEntity.property("actualDate").ge(givenDate))
            .add(AuditEntity.id().eq(givenEntityId)))
        .getSingleResult();
```

这个 最小化() 和 最大化() 方法返回一个标准, 你可以添加约束,它必须满足的实体 最大化/最小化属性。

你可能也注意到,有两个布尔参数,后通过的 创建查询。 第一个, selectEntitiesOnly ,只有有效当 你不设定一个明确的投影。 如果真的,查询的结果将 一个实体列表(变满足指定的修正 约束)。

如果错误,结果将一列三个元素的数组。 这个 第一个元素将是改变实体的实例。 第二个将一个实体 修改数据(如果没有包含自定义实体使用,这将是一个实例 的 DefaultRevisionEntity)。 第三个将的类型 修订(的值之一 RevisionType 枚举: 添加、国防部、德尔)。

第二个参数, selectDeletedEntities ,指定如果修订, 被删除的实体应该包含在结果中。 如果是的,这样的实体 将有修订,所有字段类型德尔,除了身份证, 空 。

15 7 3。 查询修改,修改给定属性的实体

对于这两种类型的查询上面描述的可以使用 特殊审计标准称为 改变了() 和 hasNotChanged() 这使得使用功能 描述 15.6 节,“跟踪实体的变化属性级” 。 他们是最适合垂直查询, 然而现有的API并不限制他们的使用水平 的人。 让我们看看以下的例子:

```
AuditQuery query = getAuditReader().createQuery()
.forRevisionsOfEntity(MyEntity.class, false, true)
.add(AuditEntity.id().eq(id));
.add(AuditEntity.property("actualDate").hasChanged())
```

这个查询将返回的所有版本MyEntity给定id、 在 actualDate 属性已更改。 使用这个查询我们不会得到所有其他修订中 actualDate 不是感动。 当然没有什么阻碍用户的结合 与一些额外的标准改变了条件——添加方法 可以用在一个正常的方式。

```
AuditQuery query = getAuditReader().createQuery()
.forEntitiesAtRevision(MyEntity.class, revisionNumber)
.add(AuditEntity.property("prop1").hasChanged())
.add(AuditEntity.property("prop2").hasNotChanged());
```

这个查询将返回水平切片为MyEntity当时 revisionNumber是生成的。 它将被限制在修订 修改 prop1 但不是 prop2 。 注意,结果集将通常也包含修订 与数字低于revisionNumber,所以我们不能读 这个查询是“给我所有MyEntities revisionNumber改变 与 prop1 修改和 prop2 不变” 。 得到这样的结果我们已经使用 forEntitiesModifiedAtRevision 查询:

```
AuditQuery query = getAuditReader().createQuery()
.forEntitiesModifiedAtRevision(MyEntity.class, revisionNumber)
.add(AuditEntity.property("prop1").hasChanged())
.add(AuditEntity.property("prop2").hasNotChanged());
```

15 7 4。 查询实体修改在一个给定的修订

基本的查询允许检索实体名称和相应的Java类改变了在一个指定的修订:

```
Set<Pair<String, Class>> modifiedEntityTypes = getAuditReader()
    .getCrossTypeRevisionChangesReader().findEntityTypes(revisionNumber);
```

(也可以从其他查询 org.hibernate.envers.CrossTypeRevisionChangesReader):

1. 列表<对象> findEntities(数字) ——返回所有审计实体的快照改变(添加、更新和删除)在一个给定的修订。 执行 n + 1 SQL查询, n 是许多不同的实体 在指定类修改修改。
2. 列表<对象> findEntities(号码,RevisionType) ——返回所有审计实体的快照变化(增加、更新或删除)在一个给定的修订 通过修改类型过滤。 执行 n + 1 SQL查询, n 是许多不同的实体类改性在指定的修订。
3. Map <,<对象> RevisionType列表> findEntitiesGroupByRevisionType(数字) ——返回一个地图包含列表按修改的实体快照操作(如。 添加、更新和删除)。 执行 3 n + 1 SQL查询, n 是许多不同的实体类改性在指定的修订。

请注意,上述方法可以合法使用只有当默认机制 启用跟踪改变实体名称(见 部分15 5 1,“跟踪实体名称修改在修正”)。

15.8。 条件审核

Envers持续审计数据在反应各种Hibernate事件(如职位、岗位插入,更新 等等),使用一系列甚至听众的 org.hibernate.envers.event 包。 默认情况下,如果Envers罐是在类路径中,事件监听器自动注册用 Hibernate。

有条件的审计可以实现覆盖一些Envers事件监听器。 使用定制的Envers事件监听器,以下步骤是必要的:

1. 关掉自动Envers事件监听器注册通过设置 hibernate.listeners.envers.autoRegister Hibernate财产 假 。
2. 为适当的事件侦听器创建子类。 例如,如果你想 有条件地审计实体插入,扩展 org.hibernate.envers.eventEnversPostInsertEventListenerImpl 类。 把条件审计逻辑在子类,叫超级方法如果 审计应该执行。
3. 创建您自己的实现 org hibernate积分器spi积分器 , 类似 org.hibernate.envers.event.EnversIntegrator 。 用你的事件 侦听器类而不是默认的。

4. 对积分器自动启动时使用Hibernate,您将需要添加一个 meta - inf / services / org hibernate积分器spi积分器 文件到你的jar。 这个文件应该包含的类的完全限定名实现接口。

15.9。 理解Envers模式

对于每个审计实体(即为每个实体包含至少一个审计字段),审核表 创建的。 默认情况下,审计表的名称是由添加一个“aud”后缀原始表名称, 但这可以被指定一个不同的后缀/前缀在配置或每个实体使用 这个 @org.hibernate.envers.AuditTable 注释。

审计表列

1. id的原始实体(这可以更多然后一列对于复合主键)
2. 修订号——一个整数。 比赛在修订的修订号实体表。
3. 修订类型——一个小整数
4. 审计领域从最初的实体

的主键的结合审计表原始id的实体和修订 数,最多有一个历史性的进入对于一个给定的实体实例在一个给定的修订。

当前的实体数据存储在原始表和在审计表。 这是一个重复的 数据,然而由于该解决方案使得查询系统更强大,内存是便宜,希望这不会是一个主要缺点为用户。 审计表中的一行与实体标识id、版本N和 数据D意味着:实体与id标识有数据D N向上修正。因此,如果我们想要找到一个实体在 修订,我们必须寻找一个行审计表的修订号小或相等 到,但尽可能大。 如果没有这样的行被发现,或一行与“删除”标志被发现,这意味着 :实体不存在,修订。

“修订类型”字段可以目前有三个值:0,1,2,这意味着添加、国防部和德尔, 分别。 一行一个修订的类型德尔只会包含实体的id和没有数据(所有 字段为空),因为它只是作为一个标志说“这个实体被删除,修改”。

此外,有一个修订实体表包含的信息 全球修订。 默认情况下生成的表命名 REVINFO 和 只包含2列: ID 和 时间戳 。 一行插入到这个表在每一个新版本,即在每个提交的事务, 改变审计数据。 这个表的名称可以配置,其列的名称以及添加 可以获得额外的列中讨论过的 15.5节,“修订日志” 。

尽管全球修订是一种很好的方式来提供正确的审计的关系,有些人指出 这可能是一个瓶颈在系统,数据非常经常修改。 一个可行的解决方案是 介绍一个选项有一个实体 “本地revisioned”,这是修正可以产生它 独立。 这不会使正确的版本控制的关系,但不会也需要 REVINFO 表。 另一个可能性是引入一个概念 “修订组”:团体的实体,分享修订编号。 每一个这样的团体必须包含一个或多个图的强连通分量引起的实体之间的关系。 你的意见是非常受欢迎的话题在论坛! :)

15.10。 生成模式与蚂蚁

如果你想生成数据库模式文件与Hibernate工具Ant任务, 你可能会发现,生成的文件不包含定义的审计 表。 生成也审计表,你只需要使用 org.hibernate.tool.ant.EnversHibernateToolTask 而不是通常的org.hibernate.tool.ant.HibernateToolTask 。 前类扩展 后者,只会增加一代版本的实体。 所以你可以使用任务 就像你以前。

例如:

```
<target name="schemaexport" depends="build-demo"
 description="Exports a generated schema to DB and file">
  <taskdef name="hibernatetool"
    classname="org.hibernate.tool.ant.EnversHibernateToolTask"
    classpathref="build.demo.classpath"/>

  <hibernatetool destdir=".">
    <classpath>
      <fileset refid="lib.hibernate" />
      <path location="${build.demo.dir}" />
      <path location="${build.main.dir}" />
    </classpath>
    <jpaconfiguration persistenceunit="ConsolePU" />
    <hbm2ddl
      drop="false"
      create="true"
      export="false"
      outputfilename="versioning-ddl.sql"
      delimiter=";"
      format="true"/>
  </hibernatetool>
</target>
```

将生成如下模式:

```
    create table Address (
        id integer generated by default as identity (start with 1),
        flatNumber integer,
        houseNumber integer,
        streetName varchar(255),
        primary key (id)
    );

    create table Address_AUD (
        id integer not null,
        REV integer not null,
        flatNumber integer,
        houseNumber integer,
        streetName varchar(255),
        REVTYPE tinyint,
        primary key (id, REV)
    );

    create table Person (
```

```
    id integer generated by default as identity (start with 1),
    name varchar(255),
    surname varchar(255),
    address_id integer,
    primary key (id)
);

create table Person_AUD (
    id integer not null,
    REV integer not null,
    name varchar(255),
    surname varchar(255),
    REVTYPE tinyint,
    address_id integer,
    primary key (id, REV)
);

create table REVINFO (
    REV integer generated by default as identity (start with 1),
    REVTSTMP bigint,
    primary key (REV)
);

alter table Person
    add constraint FK8E488775E4C3EA63
    foreign key (address_id)
    references Address;
```

15.11。 映射异常

15 11 1。 什么不是和将不支持

袋(相应的Java类型列表),因为它们可以包含非唯一元素。 原因在于坚持,例如一袋字符串年代,违反了原则 关系数据库:每个表是一组元组。 以防袋, 然而(这需要一个连接表),如果有重复的元素,两个 元组对应元素将是相同的。 Hibernate允许这个, 然而Envers(或者更确切地说:数据库连接器)将抛出一个异常 当试图坚持两个相同的元素,因为唯一约束违反。

至少有两个方法解决如果你需要袋语义:

1. 使用一个索引集合, @IndexColumn 注释,或
2. 为你提供一个惟一的id的元素 @CollectionId 注释。

15 11 2。 什么不是和 将 支持

1. 组件的集合

15 11 3。 onetomany + @JoinColumn

当一个集合映射使用这两个注释,Hibernate不 生成一个连接表。 Envers,然而,要做到这一点,所以,当你阅读 修订的相关实体已经改变,你不会得到错误的结果。

能够名额外的连接表,有一个特殊的注释: @AuditJoinTable ,有相似的语义来JPA的 @JoinTable 。

一种特殊情况是关系映射与 onetomany + @JoinColumn 在 一方面,和 @ManyToOne + (可插入的@JoinColumn = false,可更新的= false) 在许多侧。 这样的关系实际上是双向的,但拥有侧是集。

正确的审计Envers等关系,可以使用 @AuditMappedBy 注释。 它使您能够指定反向属性(使用 mappedBy 元素)。 如果 集合的索引,索引列也必须被映射在引用的实体(使用 @ column(可插入的= false,可更新的= false) ,指定使用 positionMappedBy 。 这个注释将只影响方式 Envers作品。 请注意,注释是实验和未来可能会改变。

15.12。 高级:审计表分区

15日12 1。 审计表分区的好处

因为审计表倾向于无限增长他们很快就会变成真正的大。 当审计表已经 到一定限制(不同/ RDBMS和/或操作系统)是有意义的开始使用表分区。 SQL表分区提供了许多优点,包括但并不仅限于:

1. 改善查询性能通过选择性地移动行各种分区(甚至清除旧行)
2. 更快的数据加载,创建索引,等等。

15日12 2。 合适的列审计表分区

一般SQL表必须分区在列,表内的存在。 作为一个规则是合理使用 要么 最终修订 或 端修改时间戳 柱 partioning审计表。

> **注意**
>
> 最终修订信息是不可用的AuditStrategy为默认。
> 因此以下Envers配置选项是必需的:
> org.hibernate.envers.audit_strategy =
> org.hibernate.envers.strategy.ValidityAuditStrategy
> org.hibernate.envers.audit_strategy_validity_store_revend_timestamp = 真正的
> 或者,你也可以重写默认值以下属性:
> org.hibernate.envers.audit_strategy_validity_end_rev_field_name
> org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name
> 有关更多信息,请参见 吗？？？。

为什么最后修订信息应该用于审计表partioning是基于假设 审计表应该partionioned在 “提高水平的有趣的,就像这样:

1. 几个分区与审计数据,不是很(或不再)有趣。 这可以存储在缓慢的媒体,甚至最终被净化。
2. 一些分区为审计数据,可能是有趣的。
3. 一个分区为审计数据,最有可能是有趣的。 这应该是存储在最快的媒体,对于阅读和写作。

15日12 3。 审计表分区的例子

为了确定一个合适的柱的增加水平的兴趣度”，考虑一个简化的例子,一位不愿透露姓名的工资登记机构。

目前,工资表包含以下行X:为某个人

表15.2。 工资表

年	薪资(美元)
2006年	3300年
2007年	3500年
2008年	4000年
2009年	4500年

工资为本财政年度(2010年)是未知的。 该机构要求所有注册的变化 一个财政年度的工资记录(即一个审计跟踪)。 背后的基本原理是,决策 在一个特定的日期是基于当时的注册工资。 在任何时候,它必须是可能的 繁殖的原因某些决定在一个特定的日期。

下列审计信息是可用的,排序的顺序上出现:

表15.3。 薪水——审计表

年	修订类型	修改时间戳	薪资(美元)	端修改时间戳
2006年	添加	2007-04-01	3300年	空
2007年	添加	2008-04-01	35	2008-04-02
2007年	国防部	2008-04-02	3500年	空
2008年	添加	2009-04-01	3700年	2009-07-01
2008年	国防部	2009-07-01	4100年	2010-02-01
2008年	国防部	2010-02-01	4000年	空
2009年	添加	2010-04-01	4500年	空

15日12 3 1。 确定一个合适的分区列

这个数据划分，“水平的兴趣度”必须定义。 考虑以下:

1. 2006财政年度的只有一个修订。 它有古老 修改时间戳 所有的审计行,但仍应被看作是有趣的,因为它是最新的修改 本财政年度的工资表,它的 端修改时间戳 是空的。

   也请注意,这将是非常不幸的,如果在2011年将会有一个更新的工资财政 2006年(这是可能的在直到至少10年财政年度后)和审计 信息将会被转移到一个缓慢的磁盘(基于年龄的 修改时间戳)。 记住,在这种情况下Envers必须更新 这个 端修改时间戳 最近的审计行。

2. 有两个修订在2007财政年度的工资都有几乎相同的 修改时间戳 和一个不同的 端修改时间戳 。 乍一看,很明显,第一个版本是一个错误,可能无趣的。 唯一有趣的修订为2007是一个用 端修改时间戳 NULL。

基于上述,显然,只有 端修改时间戳 适合 审计表分区。 这个 修改时间戳 是不合适的。

15日12 3 2。 确定一个合适的分区方案

一个可能的分区方案工资表如下:

1. 端修改时间戳 年= 2008

这个分区包含审计数据,不是很(或不再)有趣。

2. 端修改时间戳 年= 2009

   这个分区包含审计数据,可能是有趣的。

3. 端修改时间戳 年> = 2010或null

   这个分区包含最有趣的审计数据。

这个分区方案还包括潜在的问题的更新 端修改时间戳 ,如果连续发生的审计表被修改。 尽管Envers将更新 端修改时间戳 审计行来 系统日期在即时修改,审计行仍将在同一分区内 (“扩展桶”)。

和在2011年的时候,最后一个分区(或“扩展桶”)分为两个新分区:

1. 端修改时间戳 年= 2010

   这个分区包含审计数据,可能是有趣的(2011年)。

2. 端修改时间戳 年> = 2011或null

   这个分区包含最有趣的审计数据和新“扩展桶”。

15.13。 Envers链接

1. Hibernate主页
2. 论坛
3. JIRA问题跟踪器 (当添加Envers有关问题,一定要选择“Envers”组件!)
4. IRC频道
5. Envers博客
6. faq

第十六章。多租户

表的内容

16.1。 多租户是什么?

术语多租户一般应用于软件开发,指示一个架构中 一个应用程序的运行实例同时服务多个客户(租户)。 这是 在SaaS解决方案高度常见。 隔离信息(数据、定制等)用于修饰或说明 不同的租户是一个特别的挑战在这些系统。 这包括每个租户所拥有的数据 存储在数据库中。 正是这最后一块,有时被称为多租户数据,我们将集中。

16.2。 多租户数据的方法

有三个主要方法来隔离信息在这些多租户系统是密切相关的 与不同的数据库模式定义和JDBC设置。

> **注意**
>
> 每种方法都有优点和缺点以及特定的技巧和注意事项。 这样 主题超出了本文的范围文档。 许多资源存在,深入研究这些 其他的话题。 一个例子是 http://msdn.microsoft.com/en-us/library/aa479086.aspx 这确实一个伟大的工作,涵盖这些主题。

16 2 1。 单独的数据库

每个租户的数据保存在一个物理上独立的数据库实例。 JDBC连接会点 专门为每个数据库,因此任何池将每个租户实现一定程度的。 一个通用的应用程序方法 这里是定义一个JDBC连接池,池中每个租户实现一定程度的选择使用基于 “ 租户标识符 “ 有关当前登录用户。

16 2 2。 独立模式

每个租户的数据保存在一个不同的数据库模式在单个数据库实例。 有2 不同的方法来定义JDBC连接在这里:

- 连接可以点专门为每个模式,正如我们看到的 单独的数据库 的方法。 这是一个选择,提供 司机支持命名默认模式在连接URL或如果 池机制支持命名模式用于其连接。 使用这个 的方法,我们会有一个不同的JDBC连接池,池中每个租户实现一定程度的使用 将选定的基础上吗 ” 租户标识符 ” 相关 当前登录用户。
- 连接可以指向数据库本身(使用一些默认模式),但 连接将会改变使用SQL 设置模式 (或类似的) 命令。 使用这种方法,我们会有一个JDBC连接池使用 服务所有租户,但使用前的连接将被改变来参考 指定的模式 ” 租户标识符 ” 目前相关 登录用户。

16 2 3。 分区(鉴频器)的数据

所有数据保存在一个数据库模式。 为每个租户的数据分区的使用 分区值或鉴别器。 这种鉴频器的复杂性可能范围从一个简单的 列值到一个复杂的SQL公式。 再次,这种方法将使用一个连接池 来服务所有租户。 然而,在这种方法中,应用程序需要改变每一个 SQL语句发送到数据库的引用 ” 租户标识符 ” 鉴频器。

16.3。 多租户在Hibernate

使用Hibernate和多租户数据归结为两个一个API,然后集成块(年代)。 作为 常见的Hibernate API的努力保持简单和孤立的从任何潜在的集成的复杂性。 API只是定义通过租户标识符作为打开任何会话。

例16.1。 租户标识符指定从 SessionFactory

```
Session session = sessionFactory.withOptions()
        .tenantIdentifier( yourTenantIdentifier )
        ...
        .openSession();
```

另外,当指定配置,一个 org.hibernate.MultiTenancyStrategy 应该命名使用吗 hibernate多租户 设置。 Hibernate将执行 验证基于类型的策略你指定。 这里的战略关系隔离 上面讨论的方法。

没有

(默认)没有多租户是预期。 事实上,它被认为是一个错误,如果一个租户 标识符指定当打开一个会话使用此策略。

模式

想关联的独立模式的方法。 这是一个错误尝试打开一个会话没有 一个租户标识符使用此策略。 此外,一个 org.hibernate.service.jdbc.connections.spi.MultiTenantConnectionProvider 必须被指定。

数据库

想关联的单独的数据库方法。 这是一个错误尝试打开一个会话没有 一个租户标识符使用此策略。 此外,一个 org.hibernate.service.jdbc.connections.spi.MultiTenantConnectionProvider 必须被指定。

鉴别器

想关联的分区(鉴频器)的方法。 这是一个错误尝试打开一个 会话没有一个租户标识符使用此策略。 这个策略是尚未实现在Hibernate的4.0和4.1。 计划在5.0的支持。

16 3 1。 MultiTenantConnectionProvider

当使用要么数据库或架构方法,Hibernate需要能够获得连接 在一个特定于租户的方式。 这是所扮演的角色 org.hibernate.service.jdbc.connections.spi.MultiTenantConnectionProvider 合同。 应用程序开发人员将需要提供一个实现这个 合同。 它的大部分方法都是极其不言自明。 唯一可能不会是 getAnyConnection 和 releaseAnyConnection 。 这是 要强调指出,这些方法不接受租户标识符。 Hibernate使用这些 方法在启动期间执行各种配置,主要通过 java sql该方法 对象。

这个 MultiTenantConnectionProvider 可以指定要使用在许多 方法:

- 使用 hibernate多租户联系供应商 设置。 它可以 命名一个 MultiTenantConnectionProvider 实例, MultiTenantConnectionProvider 实现类参考或 一个 MultiTenantConnectionProvider 实现类的名字。
- 直接传递到 org.hibernate.service.ServiceRegistryBuilder 。

- 如果以上都不是选择匹配,但设置做指定一个 hibernate连接数据源 价值,Hibernate将假定它应该 使用特定的 org.hibernate.service.jdbc.connections.spi.DataSourceBasedMultiTenantConnectionProviderImpl 实现,适用于许多漂亮的合理假设当运行的内部 一个应用程序服务器,并使用一个 javax.sql.DataSource 每个租户使用。 看到它的 javadoc为更多的细节。

16 3 2。 CurrentTenantIdentifierResolver

org.hibernate.context.spi.CurrentTenantIdentifierResolver 就是合同 对于Hibernate能够解决什么应用程序认为当前租户标识符。 实现使用要么是直接传递给 配置 通过其 setCurrentTenantIdentifierResolver 法。 它也可以被指定通过 这个 hibernate租户标识符解析器 设置。

有两个情况 CurrentTenantIdentifierResolver 使用:

- 第一种情形是当应用程序正在使用 org.hibernate.context.spi.CurrentSessionContext 功能 结合多租户。 对于当前的会话功能,Hibernate将 需要打开一个会话如果它不能找到现有的一个范围。 然而,当一个会话 在多租户环境中打开租户标识符必须被指定。 这是 在 CurrentTenantIdentifierResolver 发挥作用; Hibernate将咨询实现你提供确定租户标识符来使用 当打开会话。 在这种情况下,这需要一个 CurrentTenantIdentifierResolver 被提供的。

- 其他的情况是,当你不想要显式地指定租户 标识所有的时间当我们看到 例16.1, “指定租户标识符 SessionFactory ”。 如果一个 CurrentTenantIdentifierResolver 已经指定,冬眠吗 将使用它来确定默认租户标识符使用当打开会话。

此外,如果 CurrentTenantIdentifierResolver 实现 返回 真正的 对其 validateExistingCurrentSessions 方法,Hibernate将确保任何现有会话中发现范围有一个匹配 租户标识符。 此功能只有相关当 CurrentTenantIdentifierResolver 用于当前会话设置。

16 3 3。缓存

多租户支持Hibernate能够无缝地使用Hibernate二级缓存。 关键 用于缓存数据编码租户标识符。

16 3 4。 零碎

目前模式出口不会真正使用多租户。 这可能不会改变。

JPA专家组在过程定义多租户支持即将到来的2.1 版本的规范。

16.4。 策略 MultiTenantConnectionProvider 实现者

例16.2。 MultiTenantConnectionProvider实现使用不同的连接池

```
/**
 * Simplisitc implementation for illustration purposes supporting 2 hard coded providers (pools) and levera
 * the support class {@link org.hibernate.service.jdbc.connections.spi.AbstractMultiTenantConnectionProvide
 */
public class MultiTenantConnectionProviderImpl extends AbstractMultiTenantConnectionProvider {
        private final ConnectionProvider acmeProvider = ConnectionProviderUtils.buildConnectionProvider( "a
        private final ConnectionProvider jbossProvider = ConnectionProviderUtils.buildConnectionProvider( "jb

        @Override
        protected ConnectionProvider getAnyConnectionProvider() {
                return acmeProvider;
        }

        @Override
        protected ConnectionProvider selectConnectionProvider(String tenantIdentifier) {
                if ( "acme".equals( tenantIdentifier ) ) {
                        return acmeProvider;
                }
                else if ( "jboss".equals( tenantIdentifier ) ) {
                        return jbossProvider;
                }
                throw new HibernateException( "Unknown tenant identifier" );
        }
}
```

上面的方法是有效的数据库方法。 它也是有效的模式方法 提供了底层数据库允许命名模式,连接在连接URL。

例16.3。 实现MultiTenantConnectionProvider使用单一的连接池

```
/**
 * Simplisitc implementation for illustration purposes showing a single connection pool used to serve
 * multiple schemas using "connection altering".  Here we use the T-SQL specific USE command; Oracle
 * users might use the ALTER SESSION SET SCHEMA command; etc.
 */
public class MultiTenantConnectionProviderImpl
```

```
        implements MultiTenantConnectionProvider, Stoppable {
    private final ConnectionProvider connectionProvider = ConnectionProviderUtils.buildConnectionProvid

    @Override
    public Connection getAnyConnection() throws SQLException {
        return connectionProvider.getConnection();
    }

    @Override
    public void releaseAnyConnection(Connection connection) throws SQLException {
        connectionProvider.closeConnection( connection );
    }

    @Override
    public Connection getConnection(String tenantIdentifier) throws SQLException {
        final Connection connection = getAnyConnection();
        try {
            connection.createStatement().execute( "USE " + tenanantIdentifier );
        }
        catch ( SQLException e ) {
            throw new HibernateException(
                    "Could not alter JDBC connection to specified schema [" +
                            tenantIdentifier + "]",
                    e
            );
        }
        return connection;
    }

    @Override
    public void releaseConnection(String tenantIdentifier, Connection connection) throws SQLException {
        try {
            connection.createStatement().execute( "USE master" );
        }
        catch ( SQLException e ) {
            // on error, throw an exception to make sure the connection is not returned to the pool.
            // your requirements may differ
            throw new HibernateException(
                    "Could not alter JDBC connection to specified schema [" +
                            tenantIdentifier + "]",
                    e
            );
        }
        connectionProvider.closeConnection( connection );
    }

    ...
}
```

这种方法只是相关模式的方法。

附录A。 配置属性

表的内容

背书的。 一般配置

		的类名,Hibernate org hibernate方言方言 Hibernate的 可以生成SQL优化为一个特定的关系数据库。
hibernate方言	一个完全限定类名	在大多数情况下Hibernate可以选择正确的 org hibernate方言方言 实现基于JDBC的元数据返回的JDBC驱动程序。
hibernate显示sql	真正的 或 假	写所有SQL语句到控制台。 这是一个替代设置日志类别 org hibernate sql 调试。
hibernate格式sql	真正的 或 假	的形式打印在日志和控制台的SQL。
hibernate默认模式	一个模式名称	有资格不合格表名用给定的模式或表空间中生成的SQL。
hibernate默认目录	一个目录名称	合格不合格表名与给定目录中生成的SQL。
hibernate会话工厂名称	一个JNDI名称	这个 org hibernate sessionfactory 自动绑定到JNDI中这个名字吗 被创建之后。
hibernate马克斯获取深度	一个值之间 0 和 3	设置一个最大深度为外连接获取树为单端关联。 一个单端 协会是一个一对一或者多对一的协会。 一个值的 0 禁用默认的外 连接抓取。
hibernate默认批量抓取大小	4 , 8 ,或 16	默认大小为Hibernate批量抓取的关联。
hibernate默认实体模式	动态地图 或 POJO	默认模式为实体表示为所有会话打开从这个 SessionFactory ,默认为 POJO 。
hibernate订单更新	真正的 或 假	部队Hibernate订购SQL更新的主键值的项目正在更新。 这 减少了事务死锁的可能性在高并发系统。
hibernate生成统计	真正的 或 假	使Hibernate收集统计信息对性能调优。
hibernate使用标识符回滚	真正的 或 假	如果真的,生成的标识符属性重置为默认值对象时 删除。

hibernate使用sql注释	真正的 或 假	如果真的,Hibernate生成的注释中SQL,更易于调试。

a。 数据库配置

表a . 1。 JDBC属性

财产	例子	目的
hibernate jdbc取大小	0 或整数	一个非零值决定了JDBC取大小,通过调用 Statement.setFetchSize() 。
hibernate jdbc批处理大小	一个值之间 5 和 30	一个非零值使得Hibernate使用JDBC2批量更新。
hibernate jdbc批处理版本数据	真正的 或 假	将此属性设置为 真正的 如果您的JDBC驱动程序返回正确的行数 从 executeBatch() 。 这个选项通常是安全的,但默认情况下是禁用的。 如果 启用时,Hibernate使用成批的DML自动版本数据。
hibernate jdbc工厂类	完全限定类名的工厂	选择一个自定义 org hibernate jdbc批处理程序 。 无关对于大多数 应用程序。
hibernate 使用jdbc resultset滚动	真正的 或 假	使Hibernate使用JDBC2可滚动结果集。 这个属性是唯一相关的 用户提供的JDBC连接。 否则,Hibernate使用连接元数据。
hibernate 使用jdbc流为二进制	真正的 或 假	使用流当写作或阅读 二进制 或 可序列化的 类型 或从JDBC。 这是一个系统级属性。
hibernate 使用jdbc得到生成的键	真正的 或 假	允许使用JDBC3冬眠 preparedstatement getgeneratedkeys() 到 检索本地生成的密钥后插入。 你需要JDBC3 +司机和JRE1.4 +。 禁用这个属性 如果你的驱动有问题与Hibernate标识符生成器。 默认情况下,它试图检测 司机的能力从连接元数据。

表a。 缓存属性

财产	例子	目的
hibernate缓存提供者类	完全限定类名	的类名,CacheProvider定制。
hibernate缓存使用最小的把	真正的 或 假	优化二级缓存操作来减少写道,代价是更频繁的读取。 这 是最有效的集群缓存默认为启用集群的缓存实现。
hibernate缓存使用查询缓存	真正的 或 假	使查询缓存。 你还需要设置个人查询可缓存的。
hibernate缓存使用二级缓存	真正的 或 假	完全禁用二级缓存,这是启用的 默认情况下,类中指定一个<缓存>映射。
hibernate缓存查询缓存的工厂	完全限定类名	一个自定义 QueryCache 接口。 默认的是内置的 StandardQueryCache 。
hibernate缓存区域前缀	一个字符串	一个二级缓存区域名称的前缀。
hibernate缓存使用结构化的条目	真正的 或 假	部队Hibernate将数据存储在二级缓存的在一个更可读的格式。

表由。 事务属性

财产	例子	目的
hibernate事务工厂类	一个完全限定类名	的类名,一个 TransactionFactory 使用Hibernate事务API。 这个 默认是 JDBCTransactionFactory)。
jta usertransaction	一个JNDI名称	这个 JTATransactionFactory 需要一个JNDI名称获取JTA UserTransaction从应用程序服务器。
manager_lookup_class	一个完全限定类名	的类名,一个 TransactionManagerLookup ,用于 结合jvm级别或小矿脉发生器在JTA环境。
hibernate事务冲洗完成前	真正的 或 假	使会话刷新在 完成前 阶段的 事务。 如果可能的话,使用内置和自动会话上下文管理相反。
hibernate交易自动关闭会话	真正的 或 假	使会议期间关闭 完成后 阶段的 事务。 如果可能的话,使用内置和自动会话上下文管理相反。

> **注意**
>
> 每个属性在以下表为前缀的 Hibernate。 。 它已经被删除 在表中以节省空间。

表各。 杂项属性

财产	例子	目的
当前会话上下文类	一个的 JTA , 线程 , 管理 ,或 定制类	提供一个自定义策略的范围电流 会话。
工厂类	org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory 或 org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory	选择HQL解析器实现。
查询替换	hqlLiteral = sql文字 或 hqlFunction = SQLFUNC	从令牌在Hibernate映射到SQL查询令牌,如函数或文字名称。
		验证或出口模式对数据库的DDL当 SessionFactory 是 创建的。

hbm2ddl.auto	验证，更新，创建，创建滴	与 **创建滴** ,数据库模式是下降的时候 SessionFactory 显式地关闭。
cglib优化器使用反射	真正的 或 假	如果启用,Hibernate使用CGLIB而不是运行时反射。 这是一个系统级 财产。 反射是有用的故障排除。 Hibernate总是需要CGLIB即使你禁用 优化器。 你不能设置这个属性在hibernate cfg xml。

由。 连接池属性

c3p0连接池属性

- hibernate c3p0最小尺寸
- hibernate c3p0马克斯大小
- hibernate c3p0超时
- hibernate c3p0马克斯语句

表本。 Proxool连接池属性

财产	描述
hibernate.proxool.xml	配置Proxool提供者使用一个XML文件(。 xml是附加自动)
hibernate.proxool.properties	配置Proxool提供者使用一个属性文件(。 属性是附加 自动)
hibernate.proxool.existing_pool	是否配置Proxool提供者从现有的池
hibernate.proxool.pool_alias	Proxool池别名使用。 要求。

> **注意**
>
> 在特定的配置信息的Proxool,参考Proxool文档可以从 http://proxool.sourceforge.net/ 。

附录B。 遗留Hibernate标准查询

表的内容

> **注意**
>
> 这个附录覆盖遗留Hibernate org hibernate标准 API,它 应该考虑弃用。 新发展的重点应该放在JPA javax.persistence.criteria.CriteriaQuery API。 最终, hibernate具体标准特性将移植到JPA扩展 javax.persistence.criteria.CriteriaQuery 。 在JPA api的细节,请参阅吗？？？。
> 这个信息是按原样复制从年长的Hibernate文档。

Hibernate的特色是直观的、可扩展的标准查询API。

责任。 创建一个 标准 实例

接口 org hibernate标准 代表一个查询与 一个特定的持久化类。 这个 会话 是一个工厂 标准 实例。

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

b 2。 缩小结果集

个人查询准则是一个实例的接口 org hibernate判据准则 。 类 org hibernate准则限制 定义 工厂的方法获取某些内置

准则 类型。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

限制可以逻辑地分组。

```
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();
```

有一系列的内置标准类型(限制 子类)。 最实用的允许您指定SQL直接。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sqlRestriction("lower({alias}.name) like lower(?)", "Fritz%", Hibernate.STRING) )
    .list();
```

这个 {别名} 占位符将被替换为行别名 查询的实体。

你也可以获得一个准则从一个 财产 实例。 您可以创建一个 财产 通过调用 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();
```

b 3。 排序结果

你可以命令结果使用 org hibernate标准订单 。

```
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();
```

b 4。协会

导航 协会使用 createCriteria() 你可以在相关的实体指定约束条件:

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

第二 createCriteria() 返回一个新的 实例的 标准 这是指元素的 这个 小猫 收集。

还有另一种形式,在某些情况下很有用:

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(createAlias() 不创建一个新的实例的 标准)。

小猫的收藏 猫 实例 前两个查询返回的是 不 预先过滤 的标准。 如果你想仅检索匹配的小猫 标准,你必须使用一个 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");
}
```

另外你可以操纵结果集使用左外连接:

```
List cats = session.createCriteria( Cat.class )
        .createAlias("mate", "mt", Criteria.LEFT_JOIN, Restrictions.like("mt.name", "good%") )
        .addOrder(Order.asc("mt.age"))
        .list();
```

这将返回所有的 猫 年代和伴侣的名字从“好” 他们的伴侣的年龄下令,所有的猫没有配偶。 这是有用,当需要秩序或限制在数据库中 返回之前复杂/大型结果集,并消除了许多例子 多个查询必须执行和结果联合 由java内存中。

没有这个功能,首先所有的猫没有异性伴侣的需要加载在一个查询。

第二个查询需要获取猫与伴侣的名字开始与“好”排序的配偶年龄。

第三,在内存中,列出了需要手动加入。

b 5。 动态关联获取

你可以指定关联获取语义在运行时使用 setFetchMode() 。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

这个查询将获取两个 伴侣 和 小猫 通过外部联接。 看到 吗? ? ? 为更多的信息。

b 6。组件

添加一个限制对财产的嵌入式组件,该组件属性 的名字应该返回的属性名当创建 限制 。 对象的标准应该创建在拥有实体,无法创建的组件 本身。 例如,假设 猫 有组件属性 FullName 与接头性能 FirstName 和 lastName :

```
List cats = session.createCriteria(Cat.class)
        .add(Restrictions.eq("fullName.lastName", "Cattington"))
        .list();
```

注意:这并不适用于当查询组件的集合,见下文 B部分。 7、 “集合”

b 7。收藏

当使用标准针对集合,有两种不同的情况下。 一是如果 集合包含实体(如。 <一对多/ > 或 <多对多/ >)或组件 (<复合元素/ >), 和第二个是如果集合包含标量值 (<元素/ >)。 在第一种情况下,语法上面给出的部分 B部分。 4、 “协会” 我们限制 小猫 收集。 基本上我们创建一个 标准 对象对收集 财产和限制实体或组件属性使用该实例。

对于queryng一组基本的价值观,我们仍然创建 标准 对象与收集,但引用值,我们使用特殊属性 “元素”。 对于一个索引收集,我们也可以参考索引属性使用 “指数”的特殊属性。

```
List cats = session.createCriteria(Cat.class)
        .createCriteria("nickNames")
                .add(Restrictions.eq("elements", "BadBoy"))
        .list();
```

b 8。 示例查询

类 org hibernate准则的例子 允许 你构建一个查询准则从一个给定的实例。

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

版本属性,标识符和协会被忽略。 默认情况下, null值属性被排除在外。

你可以调整的 例子 是应用。

```
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();
```

你甚至可以使用例子来地方标准在相关对象。

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

b 9。 预测、聚合和分组

类 org hibernate判据预测 是 工厂 投影 实例。 你可以申请一个 投影到查询调用 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();
```

没有明确的“集团”有必要在一个标准的查询。 某些 投影类型被定义为 分组预测，也出现在SQL 集团 条款。

别名可以分配给一个投影,投影值 中,可以引用限制或序。 这里有两种不同的方法 这样做:

```
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();
```

这个 别名() 和 作为() 方法简单的包装 投影实例,在另一个实例的别名, 投影 。 作为一种快捷方式,您可以设置一个别名当你添加投影到一个 投影列表:

```
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")
    专 家    进 阶    入 门    全文翻译                    Jame" )
翻译级别                                                ame" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

您还可以使用 Property.forName() 表达的预测:

```
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();
```

b 10。 超然的查询和子查询

这个 detachedcriteria 类允许您创建一个查询范围以外 一个会话,然后使用一个任意执行它 会话 。

```
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();
```

一个 detachedcriteria 也可以用来表达子查询。 准则 实例包括子查询可以获得通过 子查询 或 财产 。

```
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();
```

相关子查询也是可能的:

```
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();
```

多列限制的例子基于子查询:

```
DetachedCriteria sizeQuery = DetachedCriteria.forClass( Man.class )
    .setProjection( Projections.projectionList().add( Projections.property( "weight" ) )
                                                .add( Projections.property( "height" ) ) )
    .add( Restrictions.eq( "name", "John" ) );
session.createCriteria( Woman.class )
    .add( Subqueries.propertiesEq( new String[] { "weight", "height" }, sizeQuery ) )
    .list();
```

b 11。 查询通过自然标识符

对于大多数的查询,包括标准查询,查询缓存不是有效的 因为查询缓存失效发生过于频繁。 然而,有一个特殊的 类型的查询,您可以优化缓存失效算法:查找的 恒天然的关键。 在某些应用程序中,这种查询时有发生。 criteria API提供了特别规定对于这个用例。

首先,地图自然键的实体使用 <自然id > 和能够使用二级缓存。

```
<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>
```

这个功能并不打算使用实体 可变 自然键。

一旦您启用了Hibernate查询缓存, 这个 Restrictions.naturalId() 允许您使用 更高效的缓存算法。

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```