
Hibernate -关系型持久性为惯用的Java

Hibernate参考文档

[Hibernate团队](#)

[JBoss视觉设计团队](#)

4 1 11最后

版权©2004红帽公司。

[法律通知](#)

2013-03-18

表的内容

[前言](#)

1. 教程

1.1. 第1部分-第一个Hibernate应用程序

- 1.1.1. 设置
- 1.1.2. 第一节课
- 1.1.3. 映射文件
- 1 1 4. Hibernate配置
- 1 1 5. 建筑与Maven
- 1 1 6. 启动和助手
- 1 1 7. 加载和存储对象

1.2. 第2部分-映射关联

- 1.2.1. 映射Person类
- 1.2.2. 一个单向基于集合的协会
- 1.2.3. 协会工作
- 1.2.4. 值集合
- 1 2 5. 双向关联
- 1.2.6中. 工作双向链接

1.3. 第3部分- EventManager web应用程序

- 1.3.1. 写作基本servlet
- 1.3.2. 处理和渲染
- 1.3.3. 部署和测试

1.4. 总结

2. 架构

2.1. 概述

- 2.1.1. 最小建筑
- 2.1.2. 综合架构
- 2 1 3. 基本api

2.2. JMX集成

2.3. 上下文会话

3. 配置

- 3.1. 编程配置
- 3.2. 获得SessionFactory
- 3.3. JDBC连接
- 3.4. 可选配置属性

- 3 4 1. SQL方言
- 3.4.2. 外连接抓取
- 3 4 3. 二进制流

- 3.4.4. 二级和查询缓存
- 3.4.5. 查询语言替代
- 3.4.6. Hibernate统计
- 3.5. 日志
- 3.6. 实现 namingStrategy
- 3.7. 实现PersisterClassProvider
- 3.8. XML配置文件
- 3.9. Java EE应用服务器集成
 - 3-9-1. 事务策略配置
 - 3-9-2. jndi绑定 SessionFactory
 - 3-9-3. 当前会话上下文管理与JTA
 - 3-9-4. JMX部署
- 4. 持久化类
 - 4.1. 一个简单的POJO的例子
 - 以下4.4.1. 实现一个无参数的构造函数
 - 4.1.2. 提供一个标识符属性
 - 4.1.3. 喜欢不是final类(半可选)
 - 4.1.4. 声明访问器和调整器为持久字段(可选)
 - 4.2. 实现继承
 - 4.3. 实施 equals() 和 hashCode()
 - 4.4. 动态模型
 - 4.5. Tuplizers
 - 4.6. EntityNameResolvers
- 5. 基本的O / R映射
 - 5.1. 映射声明
 - 5.1.1. 实体
 - 5.1.2. 标识符
 - 5.1.3. 乐观锁定特性(可选)
 - 5.1.4. 财产
 - 5.1.5. 嵌入对象(aka组件)
 - 5-6. 继承策略
 - 5-7. 映射一个对一个,另一个许多联想
 - 5.1.8. 自然id
 - 5.1.9. 任何
 - 5-10. 属性
 - 5.1.11. 一些hbm. xml特异性
 - 5.2. Hibernate类型
 - 5.2.1. 实体和值
 - 5.2.2. 基本值类型
 - 5.2.3. 自定义值类型
 - 5.3. 映射一个类不止一次
 - 5.4. SQL引用标识符
 - 5.5. 生成的属性
 - 5.6. 柱变形金刚:读和写表达式
 - 5.7. 辅助数据库对象
- 6. 类型
 - 6.1. 值类型
 - 但是. 基本值类型
 - 6.1.2. 复合类型
 - 6.1.3. 集合类型
 - 6.2. 实体类型
 - 6.3. 类型分类的意义
 - 6.4. 定制类型
 - 6.4.1. 定制类型使用 org.hibernate.type.Type
 - 6.4.2. 定制类型使用 org.hibernate.usertype.UserType
 - 6.4.3. 定制类型使用 org.hibernate.usertype.CompositeUserType
 - 6.5. 类型注册
- 7. 集合映射
 - 7.1. 持久化集合
 - 7.2. 如何映射集合
 - 7.2.1. 收集外键
 - 7.2.2. 索引集合
 - 7.2.3. 集合的基本类型和可嵌入的对象
 - 7.3. 先进的集映射

- 7.3.1. 排序的集合
- 7.3.2. 双向关联
- 7.3.3. 双向联想索引集合
- 7.3.4. 三元关联
- 7.3.5. 使用一个鉴别器

专家 进阶 入门 全文翻译

翻译级别

8. 协会映射

8.1. 介绍

8.2. 单向关联

8.2.1. 多对一

8.2.2. 开始。一对一

8.2.3. 一对多

8.3. 单向关联与联接表

夹带了本条件8.3.1. 一对多

8.3.2. 多对一

8.3.3. 一对一

8.3.4. 多对多

8.4. 双向关联

8.4.1. 一对多或多对一的

8.4.2. 一对一

8.5. 双向联想联接表

8.5.1皆是如此。一对多或多对一的

8.5.2. 一个对一个

8.5.3. 多对多

8.6. 更复杂的关联映射

9. 组件映射

9.1. 依赖对象

9.2. 依赖对象的集合

9.3. 组件图指数

9.4. 组件标识符作为复合

9.5. 动态组件

10. 继承映射

10.1. 这三种策略

给。表每个类层次

10.1.2. 表每个子类

10.1.3. 表每个子类:使用一个鉴别器

10.1.4. 每个类层次混合表与表每个子类

10.1.5. 每个具体类一张表

10.1.6. 表使用隐式多态实现每个具体类

10.1.7. 混合隐多态性与其他继承映射

10.2. 限制

11. 处理的对象

11.1. Hibernate对象状态

11.2. 使对象持久

11.3. 加载一个对象

11.4. 查询

11.4.1. 执行查询

11.4.2. 过滤收集

11.4.3. 标准查询

11.4.4. 原生SQL查询

11.5. 修改持久化对象

11.6. 修改分离对象

11.7. 自动状态检测

11.8. 删除持久对象

11.9. 在两个不同的数据存储复制对象

11.10. 冲洗会话

11.11. 过渡持久性

11.12. 使用元数据

12. 只读实体

12.1. 让持久化实体只读

12.1.1. 实体不可变类

12.1.2. 加载持久化实体为只读

12.1.3. 只读实体加载从HQL查询/标准

- 12个1 4。做一个持久化实体只读
- 12.2。只读属性类型影响
 - 12 2 1。简单属性
 - 12 2 2。单向关联
 - 12 2 3。双向关联
- 13。事务和并发性
 - 13.1。会话和事务范围
 - 13 1 1。工作单元
 - 13 1 2。长对话
 - 13.1.3节。考虑对象的身份
 - 13 1 4。常见问题
 - 13.2。数据库事务界定
 - 13 2 1。不受管理的环境下
 - 13 2 2。使用JTA
 - 13 2 3。异常处理
 - 13 2 4。事务超时
 - 13.3。乐观并发控制
 - 13 3 1。应用版本检查
 - 13 3 2。扩展的会话和自动版本
 - 13 3 3。分离对象和自动版本
 - 13 3 4。定制自动版本
 - 13.4。悲观锁
 - 13.5。连接释放模式
- 14。拦截器和事件
 - 14.1。拦截器
 - 14.2。事件系统
 - 14.3。Hibernate声明式安全
- 15。批处理
 - 15.1。批量插入
 - 15.2。批量更新
 - 15.3。StatelessSession接口的
 - 15.4。dml风格操作
- 16。HQL:Hibernate查询语言
 - 16.1。大小写敏感性
 - 16.2。从条款的
 - 16.3。协会和连接
 - 16.4。形式的连接语法
 - 16.5。标识符属性
 - 16.6。select子句
 - 16.7。聚合函数
 - 16.8。多态查询
 - 16.9。where子句
 - 16.10。表达式
 - 16.11。order by子句
 - 16.12。该集团通过条款
 - 16.13。子查询
 - 16.14。HQL的例子
 - 16.15。批量更新和删除
 - 16.16。提示和技巧
 - 16.17。组件
 - 16.18。行值的构造函数的语法
- 17。标准查询
 - 17.1。创建一个 标准 实例
 - 17.2。缩小结果集
 - 17.3。排序结果
 - 17.4。协会
 - 17.5。动态关联获取
 - 17.6。组件
 - 17.7。集合
 - 17.8。示例查询
 - 17.9。预测、聚合和分组
 - 17.10。超然的查询和子查询
 - 17.11。查询通过自然标识符
- 18。原生SQL
 - 18.1。使用 SqlQuery

- 18 1 1. 标量查询
- 18 1 2. 实体查询
- 18 1 3. 处理协会和集合
- 18 1 4. 返回多个实体
- 18 1 5. 返回非受管实体
- 18 1 6. 处理继承
- 18 1 7. 参数
- 18.2. 命名的SQL查询
 - 18 2 1. 使用返回属性来显式地指定列/别名 名称
 - 18 2 2. 使用存储过程查询
- 18.3. 自定义的SQL创建、更新和删除
- 18.4. 定制SQL加载
- 19. 过滤数据
 - 19.1. Hibernate过滤器
- 20. 提高性能
 - 20.1. 抓取策略
 - 20 1 1. 处理懒惰协会
 - 20 1 2. 调谐获取策略
 - 20 1 3. 单端协会代理
 - 20 1 4. 初始化集合和代理
 - 20 1 5. 使用批量抓取
 - 20 1 6. 使用subselect抓取
 - 20 1 7. 获取配置文件
 - 20 1 8. 使用延迟属性获取
 - 20.2. 二级缓存
 - 20 2 1. 缓存映射
 - 20 2 2. 策略:只读
 - 20 2 3. 策略:读/写
 - 20 2 4. 策略:nonstrict读/写
 - 20 2 5. 策略:事务
 - 20 2 6. 缓存提供者/并发策略的兼容性
 - 20.3. 管理缓存
 - 20.4. 查询缓存
 - 20 4 1. 启用查询缓存
 - 20 4 2. 查询缓存区域
 - 20.5. 理解集合的性能
 - 20 5 1. 分类法
 - 20 5 2. 列表,地图,idbags和集是最有效的集合 更新
 - 20 5 3. 袋和列表是最有效的逆集合
 - 20 5 4. 一次删除
 - 20.6. 监控性能
 - 20 6 1. 监测SessionFactory
 - 20 6 2. 指标
- 21. 工具集导
 - 21.1. 自动模式生成
 - 21 1 1. 定制模式
 - 21 1 2. 运行该工具
 - 21 1 3. 属性
 - 21 1 4. 使用Ant
 - 21 1 5. 增量模式更新
 - 21 1 6. 使用Ant的增量模式更新
 - 21 1 7. 模式验证
 - 21 1 8. 使用Ant的模式验证
- 22. 额外的模块
 - 22.1. Bean验证
 - 22日1 1. 添加Bean验证
 - 22日1 2. 配置
 - 22日1 3. 抓违规
 - 22日1 4. 数据库模式
 - 22.2. Hibernate搜索
 - 22日2 1. 描述
 - 22日2 2. 整合与Hibernate注释

23. 例如:父/子

- 23.1. 一个注意集合
- 23.2. 双向一对多
- 23.3. 层叠生命周期
- 23.4. 小瀑布和 未保存的价值
- 23.5. 结论

24. 例如:Weblog应用程序

- 24.1. 持久化类
- 24.2. Hibernate映射
- 24.3. Hibernate代码

25. 例如:各种映射

- 25.1. 雇主/雇员
- 25.2. 作者/工作
- 25.3. 客户/订单/产品
- 25.4. 杂项例子映射
 - 25 4 1. “类型化” 一对一关联
 - 25 4 2. 组合键的例子
 - 25 4 3. 多对多与共享组合键属性
 - 25 4 4. 基于内容的歧视
 - 25 4 5. 协会在备用钥匙

26. 最佳实践

27. 数据库的可移植性的考虑

- 27.1. 可移植性基本
- 27.2. 方言
- 27.3. 方言决议
- 27.4. 标识符生成
- 27.5. 数据库函数
- 27.6. 类型映射

引用

表的列表

- 3.1. Hibernate JDBC属性
- 3.2. Hibernate Datasource属性
- 3.3. Hibernate配置属性
- 3.4. Hibernate JDBC和连接属性
- 3.5. Hibernate缓存属性
- 3.6. Hibernate事务属性
- 3.7. 杂项属性
- 3.8. Hibernate的SQL方言 (hibernate方言)
- 3.9. Hibernate日志类别
- 3.10. JTA TransactionManagers
- 10.1. 特性的继承映射
- 12.1. 影响房地产的只读实体类型
- 18.1. 别名注射的名字
- 20.1. 缓存提供者
- 20.2. 缓存并发策略支持
- 21.1. 总结
- 21.2. SchemaExport 命令行选项
- 21.3. SchemaExport连接属性
- 21.4. SchemaUpdate 命令行选项
- 21.5. SchemaValidator 命令行选项

列表的例子

- 4.1. 简单的POJO代表一只猫
- 4.2. 禁用代理在 hbm xml
- 4.3. 禁用代理在注释
- 4.4. 代理接口在 hbm xml
- 4.5. 代理接口的注释
- 4.6. 指定自定义tuplizers在注释
- 4.7. 指定自定义tuplizers在 hbm xml
- 5.1. @NotFound注释
- 5.2. @OnDelete注释
- 5.3. @ForeignKey注释
- 5.4. 一个对一个协会
- 6.1. 定义和注册自定义类型
- 6.2. 定义自定义UserType
- 6.3. 定义自定义CompositeUserType
- 6.4. 覆盖标准 StringType
- 6.5. 从BasicType.java片段
- 7.1. Hibernate使用自己的集合实现
- 7.2. 集合映射使用,@JoinColumn onetomany
- 7.3. 集合映射使用,@JoinTable onetomany

- 7.4. 映射一个集使用<设置>
- 7.5. 选择<一对多>元素
- 7.6. <地图>元素的映射
- 7.7. 有序列表使用 @OrderBy
- 7.8. 显式索引列使用 @OrderColumn
- 7.9. 索引列表元素在xml索引的集合 映射
- 7.10. 使用目标实体属性图主要通过 @MapKey
- 7.11. 地图的关键,基本类型使用 @MapKeyColumn
- 7.12. 地图关键xml映射元素
- 7.13. 关键的多对多映射
- 7.14. 基本类型映射的集合通过 @ElementCollection
- 7.15. @ElementCollection为嵌入对象
- 7.16. <元素>标记集合值使用映射 文件
- 7.17. 分类收集与@Sort
- 7.18. 分类收集使用xml映射
- 7.19. 排序在数据库使用类型的排序
- 7.20. 通过查询筛选排序
- 7.21. 双向一对多和多对一方作为协会 所有者
- 7.22. 双向关联有1至许多一侧 所有者
- 7.23. 双向一对多通过Hibernate映射文件
- 7.24. 通过@ManyToOne多对多关联
- 7.25. 默认值 @ManyToOne (单向)
- 7.26. 默认值 @ManyToOne (双向)
- 7.27. 多对多关联使用Hibernate映射文件
- 7.28. 效应的逆与非逆侧的许许多多 协会
- 7.29. 双向关联与索引收集
- 7.30. 双向关联与索引收集,但没有索引 列
- 7.31. 三元关联映射
- 7.32. 示例类 父 和 孩子
- 7.33. 一个到多个单向 父子 关系使用注释
- 7.34. 一个到多个单向 父子 关系使用映射文件
- 7.35. 表定义为单向 父 —— 孩子 关系
- 7.36. 一个到多个双向 父子 关系使用注释
- 7.37. 一个到多个双向 父子 关系使用映射文件
- 7.38. 表定义为双向 父 —— 孩子 关系
- 7.39. 执行非空约束在单向关系使用 注释
- 7.40. 执行非空约束在单向关系使用 映射文件
- 7.41. 许许多多 父子 关系 使用注释
- 7.42. 许许多多 父子 关系 使用映射文件
- 7.43. 表定义为许许多多relationship
- 11.1. 定义一个命名查询使用 @NamedQuery
- 11.2. 定义一个命名查询使用 <查询>
- 11.3. 参数绑定的命名查询
- 11.4. onetomany 与 orphanRemoval
- 18.1. 命名为sql查询使用 < sql查询>映射 元素
- 18.2. 执行一个已命名查询
- 18.3. 与协会命名的sql查询
- 18.4. 命名查询返回一个标量
- 18.5. < resultSet >映射用于外部化映射 信息
- 18.6. 以编程方式指定结果的映射信息
- 18.7. 命名为SQL查询使用 @NamedNativeQuery 连同 @SqlResultSetMapping
- 18.8. 隐式结果集映射
- 18.9. 在@FieldResult使用点符号用于指定关联
- 18.10. 标量值通过 @ColumnResult
- 18.11. 定制CRUD通过注释
- 18.12. 定制CRUD XML
- 18.13. 重写SQL语句的集合使用 注释
- 18.14. 重写SQL语句对于二次表
- 18.15. 存储过程和他们的返回值
- 19.1. @FilterDef和@Filter注释
- 19.2. 使用 @FilterJoinTable 对于filtering在 关联表
- 19.3. 定义一个过滤器定义通过 <过滤器def >
- 19.4. 附加一个过滤器,一个类或集合使用 <过滤器>
- 20.1. 指定一个获取配置文件使用 @FetchProfile
- 20.2. 指定一个获取配置文件使用 <获取配置文件> 外 <类> 节点
- 20.3. 指定一个获取配置文件使用 <获取配置文件> 里面 <类> 节点
- 20.4. 激活配置文件获取给定 会话
- 20.5. 定义缓存并发策略通过 @Cache
- 20.6. 缓存集合使用注释
- 20.7. @Cache 注释与 属性
- 20.8. Hibernate <缓存> 映射 元素
- 20.9. Explicitly驱逐一个缓存实例从第一级缓存 使用 会话驱逐()
- 20.10. 二级缓存回收通过 SessionFactory.evict() 和 SessionFactory.evictCollection()
- 20.11. 浏览二级缓存条目通过 统计 API
- 20.12. 使Hibernate统计
- 22.1. 使用自定义组进行验证

前言

使用面向对象的软件 and 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(和验证,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。

如果你是新的Hibernate和对象/关系映射甚至Java, 请遵循以下步骤:

1. 读 [第一章, 教程](#) 对于一个教程与循序渐进的 指令。 本教程的源代码包含在 分布 doc /引用/教程/ 目录。
2. 读 [第二章, 架构](#) 了解环境中 可以使用Hibernate。
3. 查看 例如/ 目录在Hibernate 分布。 它包含一个简单的独立的应用程序。 拷贝你的 JDBC驱动程序的 lib / 目录和编辑 etc / hibernate属性 ,指定正确的值 你的数据库。 从命令提示符分配目录, 类型 蚂蚁 (使用Ant),或在Windows、类型 构建如。
4. 使用这个参考文档作为你的主要来源 信息。 考虑阅读([JPwH](#)] 如果你需要更多的帮助与应用程序设计,或如果你喜欢一个循序渐进的指南。 还参见 <http://caveatemptor.hibernate.org> 和下载示例应用程序从[[JPwH](#)]。
5. [faq](#)是回答在Hibernate网站。
6. 链接到第三方演示、例子和教程是维护 在Hibernate网站。
7. 社区区域在Hibernate网站是一个很好的资源 设计模式和各种集成解决方案(Tomcat,JBoss AS, Struts,EJB等等)。

有很多方式来参与Hibernate社区,包括

- ▶ 尝试的东西出来,报告bug。 看到 <http://hibernate.org/issuetracker.html> 细节。
- ▶ 尝试你的手在修复一些错误或实施改进。 再次,看到 <http://hibernate.org/issuetracker.html> 细节。
- ▶ <http://hibernate.org/community.html> 列举了几种方式参与社区。
 - 有论坛用户提问和接收社区的帮助。
 - 也有 [IRC](#) 渠道对于用户和开发人员讨论。
- ▶ 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。
- ▶ 组织内传播冬眠。

第1章。教程

表的内容

1.1. 第1部分-第一个Hibernate应用程序

- 1.1.1. 设置
- 1.1.2. 第一节课
- 1.1.3. 映射文件
- 1.1.4. Hibernate配置
- 1.1.5. 建筑与Maven
- 1.1.6. 启动和助手
- 1.1.7. 加载和存储对象

1.2. 第2部分-映射关联

- 1.2.1. 映射Person类
- 1.2.2. 一个单向基于集合的协会
- 1.2.3. 协会工作
- 1.2.4. 值集合
- 1.2.5. 双向关联
- 1.2.6中. 工作双向链接

1.3. 第3部分- EventManager web应用程序

- 1.3.1. 写作基本servlet

1.3.2. 处理和渲染

1.3.3. 部署和测试

1.4. 总结

针对新用户,本章提供了一个循序渐进的介绍 冬眠,从一个简单的应用程序使用一个内存中的数据库。 这个 教程是基于早先由迈克尔Gloeg开发教程。 所有 代码是包含在 **教程/ web** 目录的项目 源。

重要

本教程希望用户有知识的两个Java和 sql。 如果你有一个有限的知识的JAVA或SQL,这是建议 那你开始有一个很好的介绍,之前的技术 去试图了解冬眠。

注意

这个发行版包含另一个示例应用程序在 这个 **教程/如** 项目源 目录。

1.1. 第1部分-第一个Hibernate应用程序

对于这个示例,我们将设置一个小型的数据库应用程序,该应用程序可以存储 我们想参加和事件有关该主机的信息(s)这些事件。

注意

尽管您可以使用任何数据库你感觉舒服的使用,我们 将使用 HSQLDB (一个内存,Java数据库)来避免描述安装/安装的任何特定的 数据库服务器。

1.1.1. 设置

我们需要做的第一件事情是设置开发环境。 我们 将使用“标准配置”所倡导的构建工具的很多这样的吗 作为 **Maven** 。 Maven,特别是,有一个 良好的资源描述这 **布局** 。 作为本教程是一个web应用程序,我们将创建和制作 使用 **src / main / java** , **src / main /资源** 和 **src / main / webapp** 目录。

我们将使用Maven在本教程中,利用它的 传递依赖管理能力以及能力的 许多ide自动为我们建立一个项目基于maven描述符。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.hibernate.tutorials</groupId>
  <artifactId>hibernate-tutorial</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>First Hibernate Tutorial</name>
  <build>
    <!-- we dont want the version to be part of the generated war file name -->
    <finalName>${artifactId}</finalName>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
    </dependency>
    <!-- Because this is a web app, we also have a dependency on the servlet api. -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
    </dependency>
    <!-- Hibernate uses slf4j for logging, for our purposes here use the simple backend -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
    </dependency>
    <!-- Hibernate gives you a choice of bytecode providers between cglib and javassist -->
    <dependency>
      <groupId>javassist</groupId>
      <artifactId>javassist</artifactId>
    </dependency>
  </dependencies>
</project>
```

提示

这不是一个要求使用Maven。 如果你希望使用其他的东西 构建本教程(如Ant),布局将保持不变。 唯一 变化是,您将需要手动占所有需要的 依赖关系。 如果你使用类似 艾薇 提供你仍依赖管理过渡的使用依赖关系 下面所提到的。 否则,你需要抓住 所有 依赖、显性和传递,并将它们添加到项目 类路径。 如果工作从Hibernate分布束,这意味着 `hibernate3.jar`,所有工件在 `lib /要求` 目录和所有文件从要么 `lib /字节码/ cglib` 或 `lib /字节码/ javassist` 目录,另外你需要两个servlet api的jar和slf4j 日志的后端。

将文件保存为 `pom.xml` 中 在项目根目录。

1.1.2. 第一节课

接下来,我们创建一个类,它代表了事件我们想商店 数据库;它是一个简单的JavaBean类和一些属性:

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

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

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

    public Date getDate() {
        return date;
    }

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

    public String getTitle() {
        return title;
    }

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

这类使用JavaBean标准命名约定财产 getter和setter方法,以及私有可见性的 字段。 虽然这是推荐的设计,它不是必需的。 Hibernate也可以直接访问字段,访问器的好处 方法是鲁棒性重构。

这个 `id` 属性包含一个独特的标识符值 对于一个特定的事件。 所有持久化实体类(有 不那么重要的从属类)将需要这样的一个标识符 如果我们想用财产完整的特性集的冬眠。 事实上, 大多数应用程序,特别是web应用程序,需要区分 对象标识符,那么你应该考虑这个功能,而 比一个限制。 但是,我们通常不会操纵的身份 的一个对象,因此setter方法应该是私人的。 只有 Hibernate 将指定的标识符,当一个对象被保存。 Hibernate可以访问 公共、私有和保护访问器方法,以及公众,私有和受保护 字段直接。 你可以选择和 您可以匹配它适合您的应用程序设计。

无参数的构造函数是一个要求所有的持久 类;Hibernate已经为您创建对象,使用Java 反射。 构造函数可以是私有的,但是包或 公众 能见度是需要运行时代理生成和有效的数据 检索没有字节码插装。

保存这个文件 `src / main / java / org / hibernate / tutorial / domain` 目录。

1.1.3. 映射文件

Hibernate需要知道如何加载和存储对象的 持久化类。 这是哪里的Hibernate映射文件 发挥作用。 映射文件告诉Hibernate 什么表 数据库必须访问,哪些列在表 它应该使用。

基本结构的映射文件如下:

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

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

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

Hibernate DTD是复杂的。您可以使用它来自动完成元素和属性的XML映射在你的编辑器或IDE。打开DTD文件在你的文本编辑器是最简单的方法得到一个概述的所有元素和属性,并查看 违约,以及一些评论。Hibernate不会加载 DTD文件从网络,但首先查的从类路径 应用程序。DTD文件包含在 **hibernate核心jar** (也包括在 **hibernate3.jar** ,如果使用分布束)。

重要

我们将省略DTD声明在未来的例子来缩短代码。它是,当然,不是可选的。

两者之间 hibernate映射 标签,包括 类 元素。所有持久化实体类(再一次,可能以后从属类,没有一流的实体的)需要一个映射到一个表的SQL数据库:

```
<hibernate-mapping package="org.hibernate.tutorial.domain">  
  
  <class name="Event" table="EVENTS">  
  
  </class>  
  
</hibernate-mapping>
```

到目前为止,我们已经告诉Hibernate如何保存和加载对象的 类 事件 表 事件。每个实例所代表的是现在 排在那张桌子。现在我们可以继续通过映射独特 标识符属性表的主键。因为我们不希望 关心处理这个标识符,我们配置Hibernate的 标识符生成策略代理主键列:

```
<hibernate-mapping package="org.hibernate.tutorial.domain">  
  
  <class name="Event" table="EVENTS">  
    <id name="id" column="EVENT_ID">  
      <generator class="native"/>  
    </id>  
  </class>  
  
</hibernate-mapping>
```

这个 id 元素的声明 标识符属性。这个 name = " id " 映射 属性声明的名字并告诉JavaBean属性 Hibernate使用 getId() 和 setId() 方法来访问属性。这个 列属性告诉Hibernate这列的 事件 表持有主键值。

嵌套的 发电机 元素指定了 标识符生成策略(即如何标识符值 生成?)。在这种情况下我们选择 原生,这提供了一个级别的可移植性根据配置的 数据库方言。Hibernate支持数据库生成的,在全球范围内 独特的,以及应用程序分配,标识符。标识符 值生成也是Hibernate的许多扩展点 你可以在你自己的策略,插件

提示

原生 不再考虑最好的策略方面的可移植性。为进一步 讨论,看到 27.4节,“标识符的一代”

最后,我们需要告诉Hibernate关于剩下的实体类 属性。默认情况下,没有属性的类被认为是 持久性:

```
<hibernate-mapping package="org.hibernate.tutorial.domain">  
  
  <class name="Event" table="EVENTS">  
    <id name="id" column="EVENT_ID">  
      <generator class="native"/>  
    </id>  
    <property name="date" type="timestamp" column="EVENT_DATE"/>  
    <property name="title"/>  
  </class>  
  
</hibernate-mapping>
```

类似 id 元素,名称 属性的 财产 元素告诉Hibernate,吸气 和setter方法来使用。在这种情况下,Hibernate将搜索 对于 获取当前日期(), 设置当前日期(), getTitle() 和 setTitle() 方法。

注意

为什么 日期 属性映射包括 列 属性,但 标题 不? 没有 列 属性,冬眠 默认情况下使用属性名 作为列名。这种方法适用于 标题,然而,日期 是一个保留 关键字在大多数数据库所以你需要将其映射到一个不同的名称。

这个 标题 映射还缺乏一个 类型 属性。这个 类型声明和使用映射文件不是Java数据类型;他们不是SQL 数据库类型要么。这些类型被称为 Hibernate映射类型,转换器可以将Java到SQL数据类型,反之亦然。再一次, Hibernate将试图确定正确的转换和映射类型本身如果 这个 类型 属性不存在的映射。在某些情况下,这 自动检测使用反射在Java类可能没有默认的你 期望或需要。就是这样的 日期 财产。Hibernate不能 知道如果这个属性,这是 java util日期,应该映射到一个 SQL 日期, 时间戳,或 时间 列。完整的日期和时间信息保存映射的属性的一个 时间戳 转换器。

提示

Hibernate使得这个映射类型确定使用反射当映射文件 是加工过的。这可能需要时间和资源,所以如果启动性能是很重要的 你应该考虑显式地定义类型使用。

保存映射文件 `src / main / 资源 / org/hibernate/tutorial/domain/event.hbm.xml` 。

1 1 4. Hibernate配置

此时,您应该已经持久化类和它的映射 文件到位。现在是时间去配置Hibernate。首先让我们来建立 HSQLDB运行在“服务器模式”

注意

我们这样做,以便数据之间仍运行。

我们将利用Maven `exec`插件启动HSQLDB服务器 通过运行: `mvn exec:java -Dexec.mainClass = " org hsqldb. 服务器" -Dexec.args = " - database. 0文件:目标/数据/教程"` 你会看到它启动和绑定到一个TCP / IP套接字,这是哪里 我们的应用程序将连接之后。如果你想开始 以全新的数据库在此教程中,关闭HSQLDB,删除 所有文件在 `目标/数据` 目录,并开始 HSQLDB再次。

Hibernate将连接到数据库的应用程序的代表,所以它需要知道 如何获得连接。对于本教程中我们将使用一个独立的连接 池(而不是一个 `javax.sql.DataSource`)。 Hibernate有 支持两个第三方开源JDBC连接池: `C3P0` 和 `proxool` 。然而,我们将使用Hibernate内置连接池对于本教程。

谨慎

内置的Hibernate连接池绝不是用于生产使用。 它 缺乏一些功能上找到什么像样的连接池。

Hibernate配置,我们可以使用一个简单的 `hibernate`属性 文件,一个 更复杂的 `hibernate cfg xml` 文件,甚至完成 编程设置。大多数用户喜欢的XML配置文件:

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

<hibernate-configuration>

  <session-factory>

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

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

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

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

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

注意

注意,这个配置文件指定一个不同的DTD

你配置Hibernate的 `SessionFactory` 。 `SessionFactory`是一个全球性的 工厂负责一个特定的数据库。如果你有几个数据库, 以便更容易 启动你应该使用一些 <会话工厂> 配置 几个配置文件。

第一个四 财产 元素包含必要的 配置JDBC连接。 方言 财产 元素指定了特定的SQL变体Hibernate生成。

提示

在大多数情况下,Hibernate是能够正确确定哪些方言使用。 看到 27.3节,“方言决议” 为更多的信息。

Hibernate的自动会话管理持久性上下文是特别有用的 在这种背景下。 这个 `hbm2ddl.auto` 选项开启自动生成 数据库模式直接进入数据库。 这还可以转变了 删除配置选项,或重定向到一个文件的帮助 这个 `SchemaExport Ant`任务。 最后,添加映射文件(年代) 对于持久化类的配置。

将文件保存为 `hibernate.cfg.xml` 到 `src/main/resources` 目录。

1 1 5. 建筑与Maven

现在我们将构建教程与Maven。 你需要 有Maven安装,它是可用的吗 [Maven下载页面](#)。 Maven将读取 `pom.xml`中 文件我们创建了 早期和知道如何执行一些基本的项目任务。 首先,允许运行 编译 目标,以确保我们可以编译 到目前为止的一切:

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

1 1 6. 启动和助手

它是时间来加载和存储一些 事件 对象,但首先你必须完成设置一些 基础设施代码。 你必须启动Hibernate的建筑 全球 `org.hibernate.sessionfactory` 对象,并将其存储在某个地方,让轻松访问应用程序代码中。 一个 `org.hibernate.sessionfactory` 用于 获得 `org.hibernate`会话 实例。 一个 `org.hibernate`会话 代表一个 单线程的工作单元。 这个 `org.hibernate.sessionfactory` 是 线程安全的全局对象实例化一次。

我们将创建一个 `HibernateUtil` helper类,负责启动和使访问 `org.hibernate.sessionfactory` 更方便。

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

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

public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

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

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

保存这个代码作为 `src/main/java/org/hibernate/tutorial/util/hibernateutil.java`

这类不仅产生了全球 `org.hibernate.sessionfactory` 参考 它的静态初始化器;它还隐藏了事实,它使用一个 静态单例。 我们也可以去查阅 `org.hibernate.sessionfactory` 参考从 JNDI在一个应用程序服务器或任何其他位置对于这个问题。

如果你给 `org.hibernate.sessionfactory` 一个名字在你的配置,Hibernate将尝试绑定它 JNDI在那个名字已经建成后。 另一个是更好的选择 使用JMX部署,让JMX能够实例化和绑定,容器 一个 `HibernateService` 到JNDI。 这样的高级选项是 稍后讨论。

你现在需要配置一个日志 系统。 Hibernate使用commons logging和提供了两个选择:Log4j和 JDK 1.4日志。 大部分开发

者更青睐于Log4j;复制 log4j.properties 从Hibernate分布 etc / 目录 你 src 目录,旁边 hibernate.cfg.xml。如果你喜欢比这更详细的输出中提供的示例配置,你可以更改设置。默认情况下,只有Hibernate启动消息显示在stdout。

本课程的基础设施已经完成,你现在准备做一些实际工作 Hibernate。

11.7. 加载和存储对象

我们现在准备开始做一些实际工作与Hibernate。让我们先写一个 EventManager 类与 main() 方法:

```
package org.hibernate.tutorial;

import org.hibernate.Session;

import java.util.*;

import org.hibernate.tutorial.domain.Event;
import org.hibernate.tutorial.util.HibernateUtil;

public class EventManager {

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

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

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

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

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

在 createAndStoreEvent() 我们创建了一个新的 事件 对象并将证据交给了Hibernate。在这一点上,Hibernate负责SQL并执行了 插入 在数据库上。

一个 org.hibernate.Session 旨在 代表的是单一的工作单元(单个原子的作品 要执行)。现在我们将保持简单和假设 一个一对一的粒度Hibernate之间 org.hibernate.Session 和一个数据库 事务。保护我们的代码从实际的基础 交易系统我们使用Hibernate org.hibernate.Session API。在这个特殊的例子中,我们使用基于jdbc的事务 语义,但它也可以运行JTA。

什么 SessionFactory.getCurrentSession() 做什么? 首先,你可以叫它多次和任何你喜欢的地方 一旦你得到你的 org.hibernate.SessionFactory。这个 getCurrentSession() 方法总是返回 “当前” 的工作单元。记住,我们交换 配置选项 对于这个机制来 “线程” 在我们的 src / main / 资源 / hibernate.cfg.xml 吗? 由于该设置,上下文的当前工作单元是绑定 当前的Java线程中执行应用程序。

重要

Hibernate提供了三种方法的当前会话跟踪。“线程”为基础的方法是不能用于生产使用;它仅仅是有用的对于这样的原型和教程。当前会话跟踪更详细的讨论 后来。

一个 org.hibernate.Session 时开始 第一个电话 getCurrentSession() 是为 当前线程。然后受Hibernate到当前 线程。当事务结束时,通过提交或 回滚,Hibernate自动unbinds了 org.hibernate.Session 从线程 和关闭它给你。如果你打电话 getCurrentSession() 再一次,你得到一个新的 org.hibernate.Session 和可以开始一个 新工作单元。

相关单位的工作范围,应该Hibernate org.hibernate.Session 被用来执行 一个或多个数据库操作? 上面的示例使用一个 org.hibernate.Session 对于一个操作。然而这是纯粹的巧合,这个例子就不复杂 足以显示任何其他方法。Hibernate的范围 org.hibernate.Session 是灵活的,但你 永远不应该设计您的应用程序使用一个新的冬眠吗 org.hibernate.Session 对于 每一 数据库操作。尽管它是 用于以下示例,考虑 会话/操作 一种反模式。一个真正的web应用程序显示在稍后的教程将 帮助说明这个。

看到 [第十三章,事务和并发](#) 为更多的信息 关于事务处理和界定。前面的 例子还跳过任何错误处理和回滚。

要运行这个,我们将利用Maven插件调用exec我们 必要的类路径设置: `mvn exec:java -Dexec.mainClass = "org.hibernate.tutorial.EventManager" -Dexec.args = "商店"`

注意

你可能需要执行 mvn 编译 第一。

您应该看到Hibernate启动,根据你的配置,大量的日志输出。接近尾声时,下面的线将显示:

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

这是 插入 执行的冬眠。

列出存储事件一个选项添加到主要的方法:

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

一个新的 listEvents()方法还补充说:

```
private List listEvents() {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();
    List result = session.createQuery("from Event").list();
    session.getTransaction().commit();
    return result;
}
```

在这里,我们使用Hibernate查询语言(HQL查询加载所有现有的 事件 数据库中的对象。 Hibernate将生成 适当的SQL,将其发送到数据库和填充 事件 对象 与数据。 您可以创建更复杂的查询与HQL。 看到 [第十六章, HQL:Hibernate查询语言](#) 为更多的信息。

现在我们可以叫我们的新功能,再使用Maven exec插件: `mvn exec:java -Dexec.mainClass = " org.hibernate教程.EventManager" -Dexec.args = "列表"`

1.2. 第2部分-映射关联

到目前为止,我们已经映射一个持久化实体类中的一个表 隔离。 让我们扩大在这一点和添加一些类协会。 我们将增加人们的应用和存储的事件列表中 他们参加。

1.2.1. 映射Person类

第一个削减的 人类如下:

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

public class Person {

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

    public Person() {}

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

保存到一个文件名 `src / main / java / org/hibernate/tutorial/domain/person.java`

接下来,创建新的映射文件 `src / main /资源/ org/hibernate/tutorial/domain/person.hbm.xml`

```
<hibernate-mapping package="org.hibernate.tutorial.domain">
  <class name="Person" table="PERSON">
    <id name="id" column="PERSON_ID">
      <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>
  </class>
</hibernate-mapping>
```

最后,添加新的映射到Hibernate的配置:

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

创建两个实体之间的一个关联这些。人 可以参与活动,与事件的参与者。设计问题 你必须处理是:方向性、多样性、和收集行为。

1.2.2. 一个单向基于集合的协会

通过添加一个事件的集合 人 类,你可以很容易地导航到一个特定的人的事件, 没有执行显式查询——通过调用人# getEvents。多值关联 在Hibernate表示的一个Java集合框架 集合,这里我们选择一个 java.util.集合 因为集合不包含重复的元素和订购 不与我们的例子:

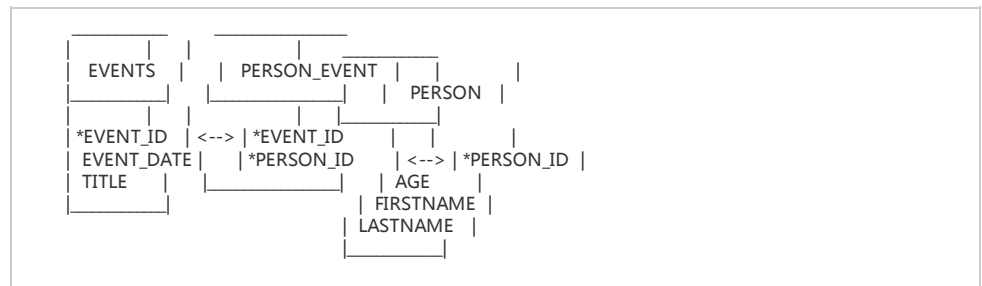
```
public class Person {  
    private Set events = new HashSet();  
  
    public Set getEvents() {  
        return events;  
    }  
  
    public void setEvents(Set events) {  
        this.events = events;  
    }  
}
```

这种关联映射之前,让我们考虑一下对方。我们可以保持这个单向或创建另一个 收集在 事件,如果我们想要的 可以从两个方向导航。这是没有必要的,从功能的角度。你总是可以执行一个显式的 查询来检索参与者对于一个特定的事件。这是一个设计的选择留给你,但什么是清晰的吗 讨论是多重性的协会:“许多”价值 在双方称为 多对多 协会。因此,我们使用Hibernate的多对多映射:

```
<class name="Person" table="PERSON">  
    <id name="id" column="PERSON_ID">  
        <generator class="native"/>  
    </id>  
    <property name="age"/>  
    <property name="firstname"/>  
    <property name="lastname"/>  
  
    <set name="events" table="PERSON_EVENT">  
        <key column="PERSON_ID"/>  
        <many-to-many column="EVENT_ID" class="Event"/>  
    </set>  
</class>
```

Hibernate支持广泛的集合的映射,一个 集 是最常见的。对于一个多对多 协会,或 n:m 实体关系,一个 关联表是必需的。在这个表的每一行代表 之间的关系的人,一个事件。表名是 declared使用 表 属性的 集 元素。标识符列名称 该协会的人身边,是定义 关键 元素,列名为事件的 侧与 列 属性的 多对多。你也必须告诉Hibernate 类的对象在您的收集(类的 另一边的收藏的引用)。

数据库模式这种映射的形式是:



1.2.3. 协会工作

现在我们将把一些人物和事件联系在一起,在新方法 EventManager :

```
private void addPersonToEvent(Long personId, Long eventId) {  
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();  
    session.beginTransaction();  
  
    Person aPerson = (Person) session.load(Person.class, personId);  
    Event anEvent = (Event) session.load(Event.class, eventId);  
    aPerson.getEvents().add(anEvent);  
  
    session.getTransaction().commit();  
}
```

在加载 人 和一个 事件,只需修改集合使用 正常的收集方法。没有显式的调用 Update() 或 save(); Hibernate自动检测到收集已被修改 和需要更新。这就是所谓的 自动脏检查。你也可以尝试 它通过修改姓名或日期你的任何财产 对象。只要他们在 持久 状态,即,绑定到一个特定的冬眠 org.hibernate会话,冬眠 监控和执行SQL的任何变化在- behind时尚。同步的过程与数据库的内存状态,通常只在最后一个工作单元,称为 冲洗。在我们的代码中,工作单元 最后提交或回滚的事务数据库。

你可以加载和事件在不同单位的人的工作。或 你可以修改一个对象之外 org.hibernate.Session, 当它不是在持续状态(如果它是持久的,这之前 状态被称为 分离)。你甚至可以 修改一个收集当它分离:

```
private void addPersonToEvent(Long personId, Long eventId) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

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

    session.getTransaction().commit();

    // End of first unit of work

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

    // Begin second unit of work

    Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
    session2.beginTransaction();
    session2.update(aPerson); // Reattachment of aPerson

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

调用 更新 使分离对象 再次被绑定到持续一个新单位工作,所以任何 修改你的同时分离可以保存到 数据库。这包括任何修改(添加/删除)你对一个集合的实体 对象。

这不是多使用在我们的例子中,但它是一个重要的概念,你可以 纳入自己的应用程序。完成这个练习通过添加一个新的行动 的主要方法 EventManager 并通过命令行调用。如果 你需要的标识一个人,一个事件- save() 方法 返回(您可能要修改一些以前的方法返回该标识符):

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

这是一个例子,两个关联之间同样重要 类:两个实体。正如前面所提到的,还有其他的 类和类型在一个典型的模型,通常是“不太重要”。一些你已经看到的,像一个 int 或 以。我们称这些类 值类型,他们的实例 靠 在一个特定的实体。实例的 这些类型没有自己的身份,也不会共享 实体之间。两个人不引用相同的 FirstName 对象,即使他们有相同的 第一个名字。值类型不能只被发现在JDK,但是 你也可以写自己从属类 比如一个 地址 或 MonetaryAmount 类。事实上,在一个Hibernate 应用程序所有的JDK类被认为是值类型。

你也可以设计一个值类型的集合。这是 从一个集合的不同概念的引用其他 实体,但看起来几乎相同的Java。

1.2.4. 值集合

让我们添加一个收集电子邮件地址的 人 实体。这将呈现为一个 java.util.Set 的 以 实例:

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

public Set getEmailAddresses() {
    return emailAddresses;
}

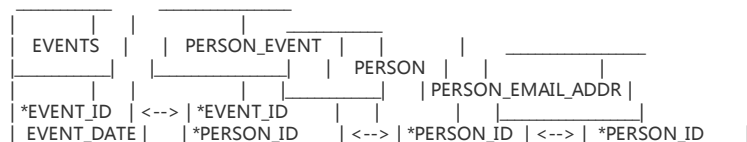
public void setEmailAddresses(Set emailAddresses) {
    this.emailAddresses = emailAddresses;
}
```

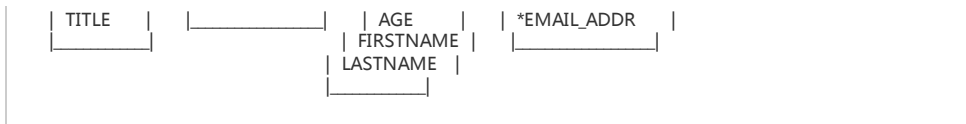
映射的 集 如下:

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

区别与早期的映射是使用 这个 元素 这告诉Hibernate的一部分 集合不包含引用到另一个实体,但是 而一个集合的元素是值类型,这里特别 类型的 字符串。小写的名字告诉你的 这是一个Hibernate映射类型/转换器。又 表 属性的 集 元素决定了表名称的集合。这个 关键 元素定义了外键列 名字在收集表。这个 列 属性 元素 元素定义了 列名称邮件地址值实际上会 被存储。

这是更新的模式:





您可以看到,主键的集合表实际上是一个复合键 使用这两列。这也意味着,不能有重复的电子邮件地址 每个人,这正是我们需要的做的语义在Java集合。

你现在可以尝试添加到这个集合的元素,就像我们之前做的 连接人与事件。它是相同的Java代码中:

```
private void addEmailToPerson(Long personId, String emailAddress) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    // adding to the emailAddress collection might trigger a lazy load of the collection
    aPerson.getEmailAddresses().add(emailAddress);

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

这次我们没有使用 取 查询 初始化集合。 监控SQL日志和尝试 优化这一渴望获取。

1 2 5. 双向关联

接下来你将地图一个双向关联。你会让 人之间的关系和事件双方的工作 在Java。数据库模式不改变,那么你仍将 有多对多的多重性。

注意

关系数据库是比网络更灵活 编程语言,它不需要一个导航 方向,数据可以查看和检索在任何可能的方式。

首先,添加一组参与者到 事件 类:

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

public Set getParticipants() {
    return participants;
}

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

现在地图这一边的协会 事件hbm xml。

```
<set name="participants" table="PERSON_EVENT" inverse="true">
  <key column="EVENT_ID"/>
  <many-to-many column="PERSON_ID" class="Person"/>
</set>
```

这些都是正常的 集 映射在两个映射文档。 注意,列名称 关键 和 多对多 在这两个映射文档交换。 最重要的是增加 逆= " true " 属性 集 元素 的事件 的集合的映射。

这意味着Hibernate应该采取其他的方面, 人 类, 当它需要找出信息之间的联系这两个。 这将是更容易 了解一旦你看到如何在两个实体之间的双向链接创建。

1.2.6中。 工作双向链接

首先,记住,Hibernate不会影响正常的Java语义。 我们如何创建一个 联系 人 和一个 事件 在单向 例子吗? 你添加的一个实例 事件 采集事件引用,的实例 人。 如果你想让这个链接 双向的,你必须做同样的另一方面通过添加 人 引用的集合 事件。 这个过程 "设置双方的联系" 与双向链接是绝对必要的。

许多开发人员程序防守和创建链接管理方法 双方设置正确(例如,在 人):

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

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

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

```

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

```

get和set方法的收集现在被保护。这允许类相同的包和子类还访问方法,但可以防止别人改变直接的集合。重复的步骤集合另一方面。

那逆映射属性? 为你和为Java,一个双向的链接是一个简单的事双方设置引用正确。冬眠,但是没有足够的信息要正确安排SQL插入和更新语句(以避免约束违反)。使一侧的协会逆告诉Hibernate考虑它镜另一边的。这是所有的,是必要的对于Hibernate来解决任何问题时出现的改变一个方向导航模型一个SQL数据库模式。规则很简单:所有的双向关联需要一边逆。在一对多关联必须多方位的,和在多对多关联您可以选择任何一方。

1.3. 第3部分- EventManager web应用程序

一个Hibernate的web应用程序使用会话和事务几乎像一个独立的应用程序。然而,一些常见的模式是很有用的。你现在可以写一个EventManagerServlet。这个servlet可以列出所有事件存储在数据库,它提供了一个HTML表单来输入新事件。

1.3.1. 写作基本servlet

首先,我们需要创建我们的基本处理servlet。因为我们的servlet只处理HTTP得到请求,我们只会实现doGet()方法:

```

package org.hibernate.tutorial.web;

// Imports

public class EventManagerServlet extends HttpServlet {

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

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

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

            // Process request and render page...

            // End unit of work
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().commit();
        }
        catch (Exception ex) {
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().rollback();
            if ( ServletException.class.isInstance( ex ) ) {
                throw ( ServletException ) ex;
            }
            else {
                throw new ServletException( ex );
            }
        }
    }
}

```

保存这个servlet作为 `src / main / java / org / hibernate / tutorial / web / eventmanagerservlet.java`

该模式应用在这里叫使用“按请求会话”模式。当一个请求击中servlet时,一个新的Hibernate会话是通过第一次调用打开getCurrentSession()在SessionFactory。数据库事务是开始。所有数据访问发生在一个事务,无论读或写数据。不使用自动提交模式在应用程序。

做不使用一个新的Hibernate会话对于每一个数据库操作。使用一个Hibernate会话这是作用域为整个请求。使用getCurrentSession(),所以它是自动绑定到当前的Java线程。

接下来,可能操作处理的请求和响应的HTML呈现。我们将去那部分很快。

最后,工作单元结束时处理和渲染完成。如果任何问题发生在处理和呈现,将会抛出一个异常和数据库事务回滚。这就完成了使用“按请求会话”模式。而不是事务界定在每个servlet的代码,你也可以写一个servlet过滤器。看到Hibernate网站和Wiki的更多信息这一模式称为公开会议针对。你需要它作为很快当你考虑渲染你的视图JSP,而不是在一个servlet。

1.3.2. 处理和渲染

现在你可以执行请求的处理和渲染的页面。

```

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

```

```

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

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

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

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

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

```

这种编码风格,混合了Java和HTML,不会规模 在一个更复杂的应用程序;记住,我们只是说明 基本概念在本教程中,冬眠 代码将输出一个HTML 页眉和页脚。 在这个页面,一个HTML表单事件条目和 一个列表的所有事件在数据库被打印。 第一个方法是琐碎的,只有输出HTML:

```

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

```

这个 listEvents() 方法使用Hibernate 会话 绑定到当前线程执行 查询:

```

private void listEvents(PrintWriter out, SimpleDateFormat dateFormatter) {

    List result = HibernateUtil.getSessionFactory()
        .getCurrentSession().createCriteria(Event.class).list();
    if (result.size() > 0) {
        out.println("<h2>Events in database:</h2>");
        out.println("<table border='1'>");
        out.println("<tr>");
        out.println("<th>Event title</th>");
        out.println("<th>Event date</th>");
        out.println("</tr>");
        Iterator it = result.iterator();
        while (it.hasNext()) {
            Event event = (Event) it.next();
            out.println("<tr>");
            out.println("<td>" + event.getTitle() + "</td>");
            out.println("<td>" + dateFormatter.format(event.getDate()) + "</td>");
            out.println("</tr>");
        }
        out.println("</table>");
    }
}

```

最后,商店 行动是被派去的 createAndStoreEvent() 方法,它也使用 这个 会话 当前线程:

```

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

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

```

servlet现在已经完成。 一个请求,servlet将被处理 在一个 会话 和 事务 。 作为 早在独立的应用程序,Hibernate可以自动绑定这些 对象当前执行的线程。 这给了你更多的自由层 你的代码和访问 SessionFactory 在任何一种你喜欢的方式。 通常你会使用更复杂的设计和移动数据访问代码 到数据访问对象(DAO模式)。 看到Hibernate Wiki更多的例子。

1.3.3. 部署和测试

要部署这个应用程序进行测试,我们必须创建一个 Web归档(战争)。 首先我们必须定义战争的描述符 作为 `src / main / webapp / - inf / web . xml`

```

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

```

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

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

构建和部署呼叫 mvn包 在你的 项目目录和复制 **hibernate教程战争** 文件到你的Tomcat **webapps** 目录。

注意

如果你没有Tomcat安装,下载它从 <http://tomcat.apache.org/> 和遵循 安装说明。我们的应用程序需要 没有改变标准Tomcat配置。

一旦部署和Tomcat运行时,访问应用程序 [http://localhost:8080 / hibernate教程/ eventmanager](http://localhost:8080/hibernate教程/eventmanager) 。 让 相信你看了Tomcat 日志看到Hibernate初始化当第一 请求到达你的servlet(静态初始化器在 HibernateUtil 被称为)和获取详细的输出如果发生任何异常。

1.4. 总结

本教程介绍了基础知识的编写一个简单的独立的Hibernate应用程序 和一个小型的web应用程序。 更多教程都可以从 [Hibernate 网站](#) 。

第二章。架构

表的内容

2.1. 概述

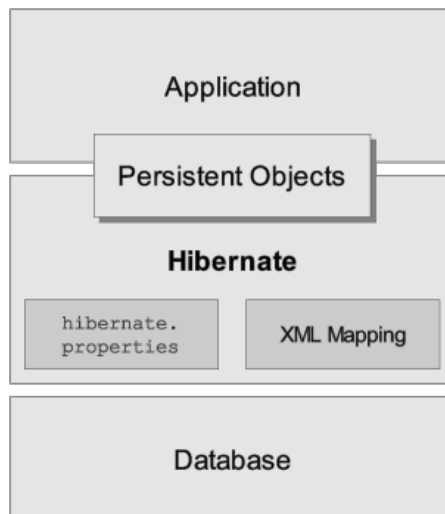
- 2.1.1. 最小建筑
- 2.1.2. 综合架构
- 2.1.3. 基本api

2.2. JMX集成

2.3. 上下文会话

2.1. 概述

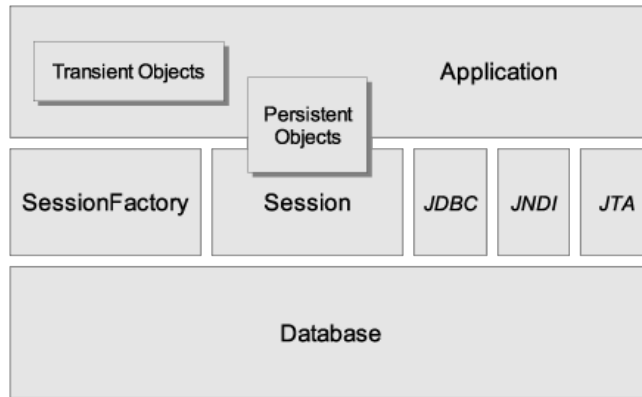
下图提供了一个大体的Hibernate架构:



不幸的是我们不能提供一个详细视图的所有可能的运行时体系结构。 Hibernate是 足够灵活,可用于多种方式在许多、许多的架构。 然而,我们将会 说明2因为它们极端特别。

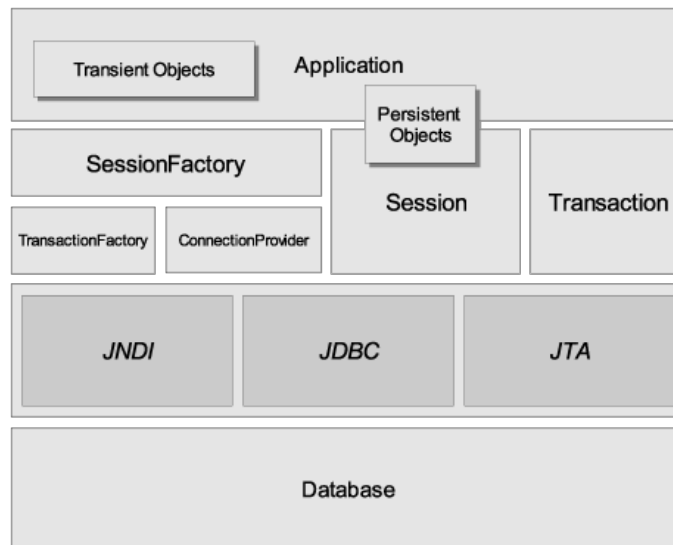
2.1.1. 最小建筑

“最小”架构的应用程序管理自己的JDBC连接,并提供这些 连接Hibernate;此外应用程序管理事务本身。 这种方法 使用 Hibernate api的最小子集。



2.1.2. 综合架构

“全面”的体系结构抽象应用远离底层JDBC / JTA api和 允许Hibernate管理细节。



2 1 3. 基本api

这里有快速讨论一些API对象描述在前面的图(你会 再看到他们在更多的细节在后面的章节)。

SessionFactory(org hibernate sessionFactory)

一个线程安全的,不可变的缓存编译映射为一个单一的数据库。 一个工厂 org hibernate会话 实例。 一个客户 的 org.hibernate.connection.ConnectionProvider 。 选择 维护一个 二级缓存 的数据的可重用的之间 事务在一个过程或 集群级别。

会话(org hibernate会话)

一个单线程的,短暂的对象代表之间的一段对话 应用程序和持久性存储。 包装JDBC java sql连接 。 工厂 org hibernate事务 。 维护一个 第一级缓存 持续应用程序的持久性对象 和集合;这个缓存时使用对象图导航或查找 对象 标识符。

持久对象和集合

短暂的,单线程的对象包含持久状态和业务 函数。 这些可以是普通javabean / pojo。 整整一个与之关联 org hibernate会话 。 一旦 org hibernate会话 是封闭的,他们会分离吗 和免费使用在任何应用程序层(例如,直接作为数 据传输对象 从演示)。 第11章, 处理的对象 讨论了瞬态, 持久和分离对象状态。

瞬态和分离对象和集合

持久化类的实例目前没有关联到一个 org.hibernate.Session 会话。他们可能被实例化 应用程序和没有坚持,或者他们可能已经实例化的 关闭 org.hibernate.Session 会话。第11章, 处理对象 讨论了瞬态、持久性和分离对象状态。

事务(org.hibernate.Transaction)

(可选)的一个单线程的,短暂的对象由应用程序使用以 指定原子工作单元。它抽象了底层JDBC应用程序, JTA或者CORBA 交易。一个 org.hibernate.Session 可能跨几个 org.hibernate.Transaction 年代在某些情况下。然而, 事务界定,要么使用底层的 API或 org.hibernate.Transaction,从来不是可选的。

ConnectionProvider(org.hibernate.connection.ConnectionProvider)

(可选)的工厂,和游泳池,JDBC连接。应用程序从它抽象 底层 javax.sql.DataSource 或 java.sql.DriverManager。这不是暴露于应用程序,但它扩展和/或实现开发人员。

TransactionFactory(org.hibernate.TransactionFactory)

(可选)的工厂 org.hibernate.Transaction 实例。这不是暴露给应用程序,但它扩展和/或 实现开发人员。

扩展接口

Hibernate提供了一系列可选的扩展接口可以实现定制 你的持久层的行为。看到的API文档的细节。

2.2. JMX集成

JMX是J2EE标准Java组件的管理。Hibernate可以管理通过 一个JMX标准服务。MBean实现中提供了地理分布: org.hibernate.jmx.HibernateService。

另一个功能可作为一个JMX服务是运行时Hibernate统计。看到 节3 4 6, “Hibernate统计” 为更多的信息。

2.3. 上下文会话

大多数应用程序使用Hibernate需要某种形式的“上下文”会话,一个给定的 会话实际上是在一个给定的上下文的范围。然而, 跨应用程序 定义什么构成一个上下文通常是不同的,不同的上下文中 定义不同的作用域概念的电流。使用Hibernate应用程序 之前 到3.0版本往往利用要么本土 ThreadLocal 的上下文会话,helper类等 HibernateUtil,或利用 第三方框架,如弹簧或微微 提供代理/基于截取上下文会话。

从版本3 0 1,Hibernate添加了 SessionFactory.getCurrentSession() 法。最初,这个假定使用 JTA 交易, JTA 事务都定义范围 和上下文的当前会话。考虑到成熟的众多独立的 JTA TransactionManager 实现,大部分,如果不是全部, 应用程序应该使用 JTA 事务管理,不管 他们被部署到一个 J2EE 集装箱。在此基础上, JTA 基于上下文的会话都需要使用。

然而,从版本3.1开始,后面的处理 SessionFactory.getCurrentSession() 现在是可插拔的。来, 结束,一个新的扩展接口, org.hibernate.context.spi.CurrentSessionContext, 和一个新的配置参数 hibernate当前会话上下文类, 已经被添加到允许 插入性的范围和上下文定义当前会话。

看到的Javadocs org.hibernate.context.spi.CurrentSessionContext 接口的详细讨论其合同。它定义了一个方法, currentSession(), 实现负责 跟踪当前上下文的会话。开箱即用的,Hibernate提供了三个 这个接口的实现:

- ▶ org.hibernate.context.internal.JTASessionContext :当前会话 跟踪和作用域由一个吗 JTA 事务。处理 这是完全一样的 老jta只有方法。看到JavaDocs 详情。
- ▶ org.hibernate.context.internal.ThreadLocalSessionContext :当前 会话跟踪的执行线程。看到Javadocs详情。
- ▶ org.hibernate.context.internal.ManagedSessionContext :当前 会话跟踪的执行线程。然而,你负责 绑定和解开一个 会话 实例和静态方法 在这类:它不开放,冲洗,或关闭 会话。

前两个实现提供一个“一个会话——一个数据库事务”编程 模型。这也是已知的和作为 使用“按请求会话”模式。一开始 和结束时,Hibernate会话被定义为一个数据库事务的持续时间。如果你使用程序性事务界定在普通市场没有JTA,建议您 使用 Hibernate 事务 API隐藏底层的交易系统 从你的代码。如果你使用JTA,您可以使用JTA事务接口来标定。如果你 执行在一个 EJB容器支持CMT、事务边界以声明的方式定义 和你不需要任何事务或会话划分操作在你的代码。指 第十三章, 事务和并行性 为更多的信息和代码示例。

这个 hibernate当前会话上下文类 配置参数 定义哪个 org.hibernate.context.spi.CurrentSessionContext 实现 应该被使用。为向后兼容性,如果这个配置参数没有设置 但一个 org.hibernate.transaction.TransactionManagerLookup 配置, Hibernate将使用 org.hibernate.context.internal.JTASessionContext。通常,该参数的值只会名称实现类 使用。对于这三个 开箱即用的实现,然而,有三个相应的 短的名字:“jta”、“线”、“管理”。

第三章. 配置

表的内容

- 3.1. 编程配置
- 3.2. 获得SessionFactory
- 3.3. JDBC连接
- 3.4. 可选配置属性

- 3 4 1. SQL方言
- 3.4.2. 外连接抓取
- 3 4 3. 二进制流

- 3.4.4. 二级和查询缓存
- 3.4.5. 查询语言替代
- 3.4.6. Hibernate统计

- 3.5. 日志
- 3.6. 实现 namingStrategy
- 3.7. 实现PersisterClassProvider
- 3.8. XML配置文件
- 3.9. Java EE应用服务器集成

- 3-9-1. 事务策略配置
- 3.9.2. jndi绑定 SessionFactory
- 3.9.3. 当前会话上下文管理与JTA
- 3.9.4. JMX部署

Hibernate是下运作而设计,在许多不同的环境中,因此,有一个广泛的配置参数。幸运的是,大多数有合理的默认值和Hibernate是分布式的例子hibernate属性文件在etc / 显示的各种选项。只是把这示例文件在您的类路径下,根据自己的需要定制它。

3.1. 编程配置

的一个实例 org.hibernate.cfg.Configuration 代表一个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.Configuration 用于构建一个不变的 org.hibernate.SessionFactory。映射编译从不同的XML映射文件。

你可以获得一个 org.hibernate.cfg.Configuration 实例的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 addResource()。例如:

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

另一种方法是指定映射的类和允许 Hibernate找到映射文档:

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

Hibernate会搜索映射文件命名 / org/hibernate/auction/item.hbm.xml 和 / org/hibernate/auction/bid.hbm.xml 在类路径中。这种方法消除了任何硬编码的文件名。

一个 org.hibernate.cfg.Configuration 还允许您指定的配置属性。例如:

```
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");
```

这不是唯一的方法,通过配置属性 Hibernate。其他选项包括:

1. 通过一个实例, java.util.Properties 到 Configuration.find()。
2. 地方文件命名 hibernate属性 在根目录的路径。
3. 集系统属性使用 Java -Dproperty =value。
4. 包括 <属性> 元素 hibernate.cfg.xml (这是稍后讨论)。

如果你想开始迅速 hibernate属性 是最简单的方法。

这个 org.hibernate.cfg.Configuration 是打算作为一个启动时间对象,一次将被丢弃 SessionFactory 被创建。

3.2. 获得SessionFactory

当所有的映射已经解析的 org.hibernate.cfg.Configuration,应用程序必须获得一个工厂吗 org.hibernate.SessionFactory 实例。这工厂的目的是由所有应用程序线程共享:

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

Hibernate应用程序确实允许实例化不止一个 org.hibernate.SessionFactory。这是有用的,如果你正在使用多个数据库。

3.3. JDBC连接

它是可取的 org.hibernate.SessionFactory 创建和池JDBC连接给你。如果采取这种方法,打开 org.hibernate.SessionFactory 一样简单为:


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

一旦你开始一项任务,需要访问数据库,一个JDBC 连接将会从池中获得。

你可以这样做之前,您首先需要通过一些JDBC连接 属性来冬眠。 所有冬眠的属性名和语义 类中定义了 org.hibernate.cfg环境。 最重要的设置JDBC连接配置进行了概述 下面。

Hibernate将获得和池连接使用 java.sql.drivermanager 如果你设置以下 属性:

表3.1. Hibernate JDBC属性

属性名	目的
hibernate连接驱动程序类	JDBC驱动程序类
hibernate连接url	JDBC URL
hibernate连接用户名	数据库用户
hibernate连接密码	数据库用户密码
hibernate连接池大小	联合的最大数量 连接

Hibernate的连接池算法,然而,相当 基本的。 它的目的是帮助你开始,是 不 用于生产系统 的,甚至是为 性能测试。 你应该使用一个第三方池为最佳 性能和稳定性。 只是取代 [hibernate连接池大小](#) 属性 连接池的具体设置。 这将关闭Hibernate的内部 池。 例如,您可能想使用c3p0。

C3P0是一个开源的JDBC连接池分布以及 Hibernate在 [自由](#) 目录。 Hibernate将使用 它的 org.hibernate.connection.C3P0ConnectionProvider 如果您设置的连接池 [hibernate c3p0.*](#) 属性。 如果你想使用 Proxool,参考包装 [hibernate属性](#) 和Hibernate网站 更多的信息。

以下是一个例子 [hibernate属性](#) 申请c3p0:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQL82Dialect
```

内部使用一个应用程序服务器,您应该几乎总是 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。 您将需要设置至少一个以下属性:

表3.2. Hibernate Datasource属性

属性名	目的
hibernate连接数据源	数据源JNDI名称
hibernate jndi url	JNDI提供者的URL (可选)
hibernate jndi类	类的JNDI InitialContextFactory (可选)
hibernate连接用户名	数据库用户 (可选)
hibernate连接密码	数据库用户密码 (可选)

这里有一个例子 [hibernate属性](#) 文件 对于一个应用程序服务器提供的JNDI数据源:

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQL82Dialect
```

JDBC连接得到一个JNDI数据源将自动 参与应用程序的容器管理的事务 服务器。

任意的连接属性可以给我们可以通过按下 "hibernate连接" 连接属性名。 例如,您可以指定一个 [字符集](#) 连接 财产使用 [hibernate连接字符集](#)。

您可以定义您自己的插件策略获得JDBC 通过实现接口的连接 org.hibernate.connection.ConnectionProvider, 并指定您的自定义实现通过 [hibernate连接provider类](#) 财产。

3.4. 可选配置属性

有许多其他属性,这些属性控制的行为 Hibernate在运行时。 都是可选的,有合理的默认 值。

警告

其中一些属性是 "系统级" 只有。 系统级属性可以设置只能通过 java -Dproperty =价值 或 [hibernate属性](#)。 他们 不能 被设置的描述的其他技术 以上。

表3.3. Hibernate配置属性

属性名	目的
	的类名,Hibernate org.hibernate方言方言 允许 Hibernate生成SQL优化的特定关系 数据库。
<code>hibernate方言</code>	<p>如。完整类名的方言</p> <p>在 大多数情况下Hibernate确实能够选择正确的 org.hibernate方言方言 实现基于 JDBC元数据 返回的JDBC驱动程序。</p>
<code>hibernate显示sql</code>	<p>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 到 调试。</p> <p>如。真正的 假</p>
<code>hibernate格式sql</code>	<p>漂亮的打印在日志和控制台的SQL。</p> <p>如。真正的 假</p>
<code>hibernate默认模式</code>	<p>有资格不合格表名与给定 模式/表空间中生成的SQL。</p> <p>如。SCHEMA_NAME</p>
<code>hibernate默认目录</code>	<p>合格不合格表名与给定目录中 生成的SQL。</p> <p>如。目录名称</p>
<code>hibernate会话工厂名称</code>	<p>这个 org.hibernate.sessionfactory 将 被自动绑定到这个名字在JNDI后 创建的。</p> <p>如。jndi /复合/名称</p>
<code>hibernate马克斯获取深度</code>	<p>设置一个最大的“深度”的外部连接获取树 单端协会(一对一,多对一的)。 一个 0 禁用默认外连接抓取。</p> <p>如。推荐值之间 0 和 3</p>
<code>hibernate默认批量抓取大小</code>	<p>设置一个默认大小为Hibernate批量抓取的 协会。</p> <p>如。推荐值 4, 8, 16</p>
<code>hibernate默认实体模式</code>	<p>设置一个默认模式为实体表示所有 会话打开从这个 SessionFactory, 默认为 POJO。</p> <p>如。动态地图 POJO</p>
<code>hibernate订单更新</code>	<p>部队Hibernate订购SQL更新的主键 价值的项目正在更新。 这将导致更少的事务死锁在高并发系统。</p> <p>如。真正的 假</p>
<code>hibernate生成统计</code>	<p>如果启用,Hibernate将收集统计信息有用 性能调优。</p> <p>如。真正的 假</p>
<code>hibernate使用标识符回滚</code>	<p>如果启用,生成的标识符属性将被重置 当对象为默认值删除。</p> <p>如。真正的 假</p>
<code>hibernate使用sql注释</code>	<p>如果打开,Hibernate将生成的注释中 SQL,更易于调试,默认 假。</p> <p>如。真正的 假</p>
<code>hibernate id新发电机映射</code>	<p>设置相关的使用时 @GeneratedValue。 这表明无论 没有新的 IdentifierGenerator 实现用于 javax.persistence.GenerationType.AUTO, javax.persistence.GenerationType.TABLE 和 javax.persistence.GenerationType.SEQUENCE。 默认 假 保持向后 兼容性。</p> <p>如。真正的 假</p>

注意

我们建议所有新项目,利用使用 @GeneratedValue 也设置 hibernate id新发电机映射= true 作为新 发电机更高效、更接近JPA 2规范 语义。然而他们不向后兼容现有的 数据库(如果一个序列或一个表用于id生成)。

表3.4. Hibernate JDBC和连接属性

属性名	目的
<code>hibernate jdbc取大小</code>	<p>一个非零值决定了JDBC取大小(调用 Statement.setFetchSize())。</p>
<code>hibernate jdbc批处理大小</code>	<p>一个非零值允许使用JDBC2批量更新的 Hibernate。</p> <p>如。推荐值之间 5 和 30</p>
<code>hibernate jdbc批处理版本数据</code>	<p>将此属性设置为 真正的 如果您的JDBC 司机返回正确的行数 executeBatch()。 它通常是安全的把这 选项。 Hibernate会用成批的DML自动 版本数据。 默认为 假。</p> <p>如。真正的 假</p>

hibernate jdbc工厂类	<p>选择一个自定义 org.hibernate.jdbc批处理程序。大多数应用程序不需要这个配置属性。</p> <p>如。 classname.of.BatcherFactory</p>
hibernate使用jdbcresultset滚动	<p>支持使用可滚动结果集由Hibernate JDBC2。这个属性只需要在使用用户提供的JDBC连接。Hibernate使用连接元数据否则。</p> <p>如。 真正的 假</p>
hibernate使用jdbc流为二进制	<p>使用流当写/读 二进制 或 可序列化的 类型/从JDBC。 *系统级属性*</p> <p>如。 真正的 假</p>
hibernate使用jdbc得到生成的键	<p>允许使用JDBC3 preparedstatement getgeneratedkeys() 到 检索本地生成的密钥后插入。需要JDBC3 + 司机和JRE1.4 +, 设置为假如果你的驱动有问题 Hibernate标识符生成器。默认情况下,它尝试 确定使用连接元数据驱动能力。</p> <p>如。 真 假</p>
hibernate连接provider类	<p>的类名,一个自定义的 org.hibernate.connection.ConnectionProvider 它提供了JDBC连接Hibernate。</p> <p>如。 classname.of.ConnectionProvider</p>
hibernate连接隔离	<p>设置JDBC事务隔离级别。检查 java sql连接 对于有意义的 值,但注意,大多数数据库不支持所有的隔离 水平和一些定义额外的、非标分解动作。</p> <p>如。 1、2、4、8</p>
hibernate连接自动提交	<p>对于JDBC连接池允许自动提交(它不是 推荐)。</p> <p>如。 真正的 假</p>
hibernate连接释放模式	<p>指定当Hibernate应该释放JDBC连接。默认情况下,一个JDBC连接到会话举行 显式地关闭或断开连接。对于一个应用程序服务器JTA 数据源,使用 在声明中 积极 释放连接在每个JDBC调用。对于一个非jta 连接,是非常有意义的释放连接 每笔交易结束时,通过使用 交易后 。 汽车 将 选择 在声明中 对于JTA和CMT 事务策略和 交易后 JDBC事务策略。</p> <p>如。 汽车 (默认) 在近 交易后 在声明中</p> <p>这个设置 只会影响 会话 年代回来 sessionFactory.openSession 。 对于 会话 年代通过 sessionFactory.getCurrentSession , CurrentSessionContext 实现配置 使用控件连接释放模式对于那些会话 年代。看到 2.3节, "上下文会话"</p>
hibernate连接。 <返回一个只读字符串>	<p>通过JDBC属性 <返回一个只读字符串> 到 DriverManager getConnection() 。</p>
hibernatejndi。 <返回一个只读字符串>	<p>通过产权 <返回一个只读字符串> 到JNDI InitialContextFactory 。</p>

表3.5。 Hibernate缓存属性

属性名	目的
hibernate缓存提供者类	<p>的类名,一个自定义的 CacheProvider 。</p> <p>如。 classname.of.CacheProvider</p>
hibernate缓存使用最小的把	<p>优化二级缓存操作来减少写道, 代价是更频繁的读取。 这个设置是最有用的 集群的缓存,在Hibernate3,默认情况下是启用的 集群缓存实现。</p> <p>如。 真 假</p>
hibernate缓存使用查询缓存	<p>使查询缓存。 单个查询仍然需要 设置可缓存的。</p> <p>如。 真 假</p>
hibernate缓存使用二级缓存	<p>可以用来完全禁用二级缓存, 这是默认启动类,指定一个吗 <缓存> 映射。</p> <p>如。 真 假</p>
hibernate缓存查询缓存的工厂	<p>的类名,一个自定义的 QueryCache 界面,默认为内置的 StandardQueryCache 。</p> <p>如。 classname.of.QueryCache</p>
hibernate缓存区域前缀	<p>前缀用于第二级缓存区域名称。</p> <p>如。 前缀</p>
hibernate缓存使用结构化的条目	<p>部队Hibernate来存储数据的二级缓存中 更人性化的格式。</p> <p>如。 真 假</p>

hibernate缓存默认缓存并发策略 设置用于给默认的名称 org.hibernate.annotations.CacheConcurrencyStrategy 使用当要么 @Cacheable 或 @Cache 是使用。 @Cache(策略= ". . .") 是用来覆盖这个吗 默认的。

表3.6。 Hibernate事务属性

属性名	目的
hibernate事务工厂类	<p>的类名,一个 TransactionFactory 到 使用Hibernate 事务 API(默认为 JDBCTransactionFactory)。</p> <p>如。 classname.of.TransactionFactory</p>
jta usertransaction	<p>使用的JNDI名称 JTAUserTransaction 获得JTA UserTransaction 从应用程序服务器。</p> <p>如。 jndi /复合/名称</p>
manager_lookup_class	<p>的类名,一个 TransactionManagerLookup 。 它需要 jvm级别启用高速缓存或当使用小矿脉发生器在JTA 环境。</p> <p>如。 classname.of.TransactionManagerLookup</p>
hibernate事务冲洗完成前	<p>如果启用,会话将自动刷新 在完成前阶段的事务。 内置 和自动会话上下文管理优先,看到 2.3节,“上下文会话” 。</p> <p>如。 真正的 假</p>
hibernate交易自动关闭会话	<p>如果启用,会话将被自动关闭期间 完成后的阶段的事务。 内置和 自动会话上下文管理优先,看到 2.3节,“上下文会话” 。</p> <p>如。 真正的 假</p>

表3.7。 杂项属性

属性名	目的
hibernate当前会话上下文类	<p>提供一个自定义的策略范围的“当前”会话 。 看到 2.3节,“上下文会话” 为更多的信息 关于内置的策略。</p> <p>如。 JTA 线程 管理 定制类</p>
hibernate查询工厂类	<p>选择HQL解析器实现。</p> <p>如。 org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory 或 org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory</p>
hibernate查询替换	<p>用于从令牌在Hibernate映射到SQL查询 令牌(令牌可能是函数或文字名称,例如)。</p> <p>如。 hqlLiteral = sql文字,SQLFUNC hqlFunction =</p>
auto	<p>自动验证或出口模式的DDL 数据库当 SessionFactory 被创建。 与 创建滴、数据库模式将 下降当 SessionFactory 关闭 明确。</p> <p>如。 验证 更新 创建 创建滴</p>
hibernate.hbm2ddl.import_files	<p>逗号分隔的可选文件名称 包含SQL DML语句期间执行 SessionFactory 创建。 这是有用的 测试或演示:通过添加INSERT语句 例如你 可以填充你的数据库和尽量少的一组数据吗 部署。</p> <p>文件顺序问题,报表,给 文件是执行之前执行语句的以下文件。 这些语句只执行如果模式是创造了ie如果 auto 设置为 创建 或 创建滴 。</p> <p>如。 /人类sql /狗sql</p>
hibernate.hbm2ddl.import_files_sql_extractor	<p>的类名,一个自定义的 ImportSqlCommandExtractor (默认为内置的 SingleLineSqlCommandExtractor)。 这是有用的为实现专用的解析器提取 单个SQL语句从每个导入文件。 Hibernate提供了 也 MultipleLinesSqlCommandExtractor 哪一个 支持指令/评论和引用字符串蔓延 多行(强制性分号每年年底的语句)。</p> <p>如。 classname.of.ImportSqlCommandExtractor</p>
hibernate字节码使用反射优化器	<p>允许使用字节码操作代替 运行时反射。 这是一个系统级属性,不能 设置在 hibernate cfg xml 。 反射可以 有时是有用的故障排除。 Hibernate总是 需要CGLIB或javassist即使你关掉 优化器。</p> <p>如。 真正的 假</p>
hibernate字节码提供者	<p>两个javassist和cglib可以作为字节 操作引擎,缺省值是 javassist 。</p> <p>如。 javassist CGLIB</p>

3 4 1. SQL方言

总是设置 hibernate方言 财产 正确的 org.hibernate方言方言 子类 对于您的数据库。 如果你指定一个方言,Hibernate将使用明智的 默认值为其他上面列出的属性。 这意味着 你将不必手动指定它们。

表3.8. Hibernate的SQL方言 (hibernate方言)

rdbms	方言
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL 8.1	org.hibernate.dialect.PostgreSQL81Dialect
PostgreSQL 8.2 and later	org.hibernate.dialect.PostgreSQL82Dialect
MySQL5	org.hibernate.dialect.MySQL5Dialect
MySQL5 with InnoDB	org.hibernate.dialect.MySQL5InnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10 g	org.hibernate.dialect.Oracle10gDialect
Oracle 11 g	org.hibernate.dialect.Oracle10gDialect
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
Microsoft SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
SAP DB	org.hibernate.dialect.SAPDBDialect
informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
H2数据库	org.hibernate.dialect.H2Dialect
安格尔	org.hibernate.dialect.IngresDialect
进步	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
包括	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
火鸟	org.hibernate.dialect.FirebirdDialect

3.4.2. 外连接抓取

如果您的数据库支持ANSI,Oracle或Sybase风格外 连接, 外连接抓取 往往会增加 性能通过限制数量的往返和从 数据库。 这是,但是,代价是可能更多的执行工作 由数据库本身。 外连接抓取允许一个整体的图 对象连接多对一、一对多、多对多的和 一对一关联检索在单个SQL 选择。

外连接抓取可以被禁用 全球 通过设置属性 hibernate马克斯获取深度 到 0。 一个设置的 1 或更高 使外连接抓取对于一对一和多对一关联 被映射与 fetch = “加入”。

看到 20.1节, “抓取策略” 更多 信息。

3 4 3. 二进制流

甲骨文大小的限制 字节 数组,可以 被传递给和/或从它的JDBC驱动程序。 如果你想使用大 实例的 二进制 或 可序列化的 式, 您可以启用 hibernate使用jdbc流为二进制。 这是一个系统级的设置只有。

3.4.4. 二级和查询缓存

前缀的属性 hibernate缓存 允许您使用一个过程或集群级二级缓存系统 与Hibernate。 看到 20.2节, “二级缓存” 更多 信息。

3 4 5. 查询语言替代

你可以定义新的Hibernate查询令牌使用 `hibernate.query.substitutions` 查询替换。例如:

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

这将导致令牌 `true` 和 `false` 被翻译成整数中的文本生成的SQL。

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

这将允许您重新命名SQL `lower` 函数。

3.4.6. Hibernate统计

如果您启用 `hibernate.generateStatistics` , Hibernate公开了一个数量的指标优化时非常有用。运行系统通过 `SessionFactory.getStatistics()` 。 Hibernate甚至可以被配置成通过JMX公开这些数据。 读Javadoc的接口 `org.hibernate.stat.Statistics` 为更多的信息。

3.5. 日志

重要

完全过时了。 Hibernate使用JBoss日志开始在4.0。 这将被记录下来作为我们迁移这个内容开发者指南。

Hibernate利用 [简单的日志记录 立面为Java \(SLF4J\)](#)为了日志系统的各种事件。 SLF4J可以直接你的日志输出到日志框架 (NOP、几、简单、log4j版本1.2、JDK 1.4日志、JCL和logback)取决于你选择的绑定。 为了设置日志记录你需要 `slf4j-api.jar` 在您的类路径下连同jar 为你喜欢的绑定文件, `slf4j-log4j12.jar` 对于Log4J。 看到SLF4J [文档](#) 更多细节。 使用Log4j您还需要放一个 `log4j.properties` 文件在您的类路径下。 一个例子 属性文件是与Hibernate的分布 `src/main/resources` 目录。

建议您熟悉Hibernate的日志 消息。 很多工作已经投入使Hibernate日志 尽可能详细,没有让它不可读。 它是一个重要 故障诊断装置。 最有趣的日志类别是 以下:

表3.9. Hibernate日志类别

类别	函数
<code>org.hibernate.sql</code>	记录所有SQL DML语句执行时
<code>org.hibernate.type</code>	记录所有JDBC参数
<code>org.hibernate.tool.hbm2ddl</code>	记录所有SQL DDL语句执行时
<code>org.hibernate.pretty</code>	日志状态的所有实体(max 20实体)相关 与会话在冲洗时间
<code>org.hibernate.cache</code>	记录所有二级缓存的活动
<code>org.hibernate.transaction</code>	日志事务相关的活动
<code>org.hibernate.jdbc</code>	记录所有JDBC资源获取
<code>org.hibernate.hql.internal.ast</code>	日志HQL和SQL——在查询解析
<code>org.hibernate.security</code>	记录所有JAAS授权请求
<code>org.hibernate</code>	记录一切。 这是一个很多的信息,但它是 用于故障诊断

在开发应用程序时使用Hibernate,你应该差不多 永远工作在 `org.hibernate.sql` 调试 启用类别的 `org.hibernate.pretty` ,或者,或者,属性 `hibernate.show_sql` 启用。

3.6. 实现 namingStrategy

接口 `org.hibernate.cfg.NamingStrategy` 允许你指定一个“命名标准”对数据库对象和模式 元素。

你可以提供规则的自动生成数据库 标识符从Java标识符或处理“逻辑”列和 表中所给的名字映射文件为“物理”表和列 的名字。 这个特性有助于减少冗长的映射文档, 消除重复的噪声(:TBL_ 前缀, 例)。 默认策略使用Hibernate非常最小。

你可以指定一个不同的策略通过调用 `Configuration.setNamingStrategy()` 之前添加 映射:

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

`org.hibernate.cfg.ImprovedNamingStrategy` 是 内置策略,可能是一个有用的起点,一些 应用程序。

3.7. 实现PersisterClassProvider

您可以配置持续程序实现用于保存您的 实体和集合:

- ▶ 默认情况下,Hibernate使用原先持续存在的意义在 关系模型和遵循Java持久性的规范
- ▶ 您可以定义一个 PersisterClassProvider 实现,提供了实现类使用一个给定的 实体或集合
- ▶ 最后,您可以覆盖在每个实体和集合 根据映射使用 @Persister 或其 XML等价

后者在列表中更高的优先。

你可以通过 PersisterClassProvider 实例到 配置 对象。

```
SessionFactory sf = new Configuration()
    .setPersisterClassProvider(customPersisterClassProvider)
    .addAnnotatedClass(Order.class)
    .buildSessionFactory();
```

持续程序类供应商的方法,当返回一个非零 的实现类,覆盖默认的Hibernate原先持续存在。 实体 名称或集合的作用是传递给方法。 这是一个很好的方式 集中压倒一切的逻辑代替分散的原先持续存在 他们在每个实体或集合映射。

3.8. XML配置文件

另一种方法是指定一个完整的配置 配置文件中命名 hibernate cfg xml 。 这 文件可以作为替代 hibernate属性 文件或,如果两个都在场,来 覆盖属性。

XML配置文件是默认情况下将在根 你的 类路径 。 这里是一个例子:

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

<hibernate-configuration>

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

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

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

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

  </session-factory>

</hibernate-configuration>
```

这种方法的优点是外化的映射 配置文件的名称。 这个 hibernate cfg xml 是 更方便一旦你必须调整Hibernate缓存。 这是你的 选择既能使用 hibernate属性 或 hibernate cfg xml 。 都是等价的,除了 上述好处,使用XML语法。

XML配置,然后开始冬眠一样简单 为:

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

你可以选择一个不同的XML配置文件使用:

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

3.9. Java EE应用服务器集成

Hibernate有以下集成点对于J2EE 基础设施:

- ▶ 容器管理数据源 :冬眠 可以使用JDBC连接由容器进行管理和提供了通过吗 JNDI。 通常,一个JTA兼容 transactionManager 和一个 ResourceManager 照顾事务 管理(CMT),特别是分布式事务处理跨 几个数据源。 你也可以限定事务边界 以编程方式(BMT),或者你可能想要使用可选的 Hibernate 事务 保持你的API 代码便携式。
- ▶ 自动JNDI绑定 :Hibernate可以绑定 它的 SessionFactory 启动后到JNDI。
- ▶ JTA会话绑定: Hibernate 会话 可以自动绑定到范围的 JTA事务。 简单地查找 SessionFactory 从JNDI和获得当前 会话 。 让 Hibernate管理冲洗和关闭 会话 当你的JTA事务完成。 事务界定要么是 声明(CMT)或编程(BMT / UserTransaction)。
- ▶ JMX部署: 如果你有一个JMX能力 应用程序服务器(例如,JBoss AS),你可以选择部署Hibernate 作为一个管理的 MBean。 这样就可以节省你的一行启动代码来构建 你 SessionFactory 从 配置 。 容器将启动你的 HibernateService 和也照顾服务 依赖性(数据源必须是可用的,Hibernate的开始之前 等)。

取决于您的环境,您可能必须设置 配置选项 hibernate连接积极发布 如果你真正的 应用程序服务器显示“连接遏制”异常。

3 - 9 - 1. 事务策略配置

Hibernate 会话 API是独立于任何 事务界定系统在您的体系结构。 如果你让 Hibernate使用JDBC直接通过一个连接池,你可以开始和 结束你的交易通过调用JDBC API。 如果你运行在一个J2EE 应用程序服务器,您可能想要使用bean管理的事务和 调用JTA API和 UserTransaction 当 需要。

保持你的代码移植到不同这两个(和其他) 我们建议可选的Hibernate的环境 事务 API,它封装和隐藏了底层 系统。 你必须指定 一个工厂类 事务 实例通过设置Hibernate 配置属性 hibernate事务工厂类。

有三个标准,或内置,选择:

org.hibernate.transaction.JDBCTransactionFactory

代表数据库(JDBC)事务(默认)

org.hibernate.transaction.JTATransactionFactory

代表容器管理的事务,如果现有的 交易正在进行中在这个上下文(例如EJB会话 bean方法)。 否则,启动了一个新事务和 使用bean管理的事务。

org.hibernate.transaction.CMTTTransactionFactory

代表容器管理的JTA事务

您也可以定义您自己的交易策略(CORBA 事务服务,例如)。

一些功能在Hibernate(即。 、二级缓存,上下文的会话的JTA等等)需要访问JTA transactionManager 在托管环境。 在一个 应用程序服务器,因为J2EE不规范一个机制, 你必须指定如何Hibernate应获得一个参考 transactionManager :

表3.10. JTA TransactionManagers

事务工厂	应用服务器
org.hibernate.transaction.JBossTransactionManagerLookup	JBoss AS
org.hibernate.transaction.WeblogicTransactionManagerLookup	weblogic
org.hibernate.transaction.WebSphereTransactionManagerLookup	WebSphere
org.hibernate.transaction.WebSphereExtendedJTATransactionLookup	WebSphere 6
org.hibernate.transaction.OrionTransactionManagerLookup	猎户座
org.hibernate.transaction.ResinTransactionManagerLookup	树脂
org.hibernate.transaction.JOTMTransactionManagerLookup	JOTM
org.hibernate.transaction.JOnASTransactionManagerLookup	乔纳斯
org.hibernate.transaction.JRun4TransactionManagerLookup	JRun4
org.hibernate.transaction.BESTransactionManagerLookup	宝蓝ES
org.hibernate.transaction.JBossTSStandaloneTransactionManagerLookup	JBoss TS独立使用(即。 外 JBoss AS和 JNDI环境一般)。 已知的工作 org.jboss.jbosssts:jbossjta:4 11 0最终

3 9 2. jndi绑定 SessionFactory

一个jndi绑定Hibernate SessionFactory 可以 简化查找函数的工厂和创建新的 会话 年代。这不是,然而,相关的JNDI 绑定 数据源,都只是使用相同的 注册表。

如果你希望有 SessionFactory 绑定到 一个JNDI名称空间,指定一个名称(如。 java:hibernate / SessionFactory)使用属性 hibernate会话工厂名称 。 如果这个属性是 省略了, SessionFactory 将不会被绑定到吗 JNDI。 这是特别有用,在环境与一个 只读的JNDI 默认实现(在Tomcat,例如)。

当绑定 SessionFactory 到JNDI, Hibernate将使用的价值 hibernate jndi url, hibernate jndi类 实例化一个初始 上下文。 如果不指定,默认 InitialContext 将被使用。

Hibernate会自动把 SessionFactory 在JNDI在你调用 cfg.buildSessionFactory() 。 这意味着你会有 这叫在一些启动代码, 或者在应用程序、实用工具类 除非你使用JMX部署 HibernateService (这是后来在大讨论 细节)。

如果你使用一个JNDI SessionFactory、EJB或任何其他类,您可以获得 SessionFactory 使用 一个JNDI查找。

建议您绑定 SessionFactory 到JNDI在托管环境,使用一个 静态 单例否则。保护你的 应用程序代码从这些细节,我们也建议 隐藏 实际查找代码为 SessionFactory 在一个助手 类,如 HibernateUtil.getSessionFactory()。请注意,这样的类也是一个方便的方式来启动hibernate看到 第1章。

3 9 3. 当前会话上下文管理与JTA

最简单的方法来处理 会话 和 事务是Hibernate的自动“当前”会话 管理。讨论语境 会话看到 2.3节,“上下文会话”。使用 “jta” 会话上下文,如果没有冬眠 会话 与当前的JTA事务相关联,你将开始和与之相关的JTA事务第一 时间你打电话 SessionFactory.getCurrentSession()。这个 会话 年代检索通过 getCurrentSession() 在 “jta” 背景设置为自动冲洗的 事务完成前,在事务完成后关闭,并积极地释放JDBC 连接在每个语句。这允许 会话 年代管理生命周期的JTA 它关联的事务,保持清洁的用户代码等 管理问题。你的代码可以使用JTA编程方式 通过 UserTransaction 或(推荐用于便携式 代码)使用 Hibernate 事务 API来设置 事务边界。如果你运行在EJB容器,声明 事务界定与CMT是首选。

3 9 4. JMX部署

线 cfg.buildSessionFactory() 还需要 被执行的地方获得 SessionFactory 进 JNDI。你可以在一个 静态 初始化 块,就像一个在 HibernateUtil,或者你可以 部署Hibernate作为 托管服务。

Hibernate分布与 org.hibernate.jmx.HibernateService 为部署在 应用服务器与JMX功能,如JBoss as。这个 实际部署和配置特定于供应商的。这是一个 例子 jboss服务的xml 对JBoss 4 0 x:

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

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

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

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

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

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

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

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

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

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

</mbean>

</server>
```

这个文件是部署在一个目录,名为 meta - inf 和打包在一个JAR文件 扩展 sar (服务存档)。你还需要 包Hibernate,其所需的 第三方库,你的编译 持久化类,以及你的映射文件在同一存档。你的企业bean(通常会话bean)可以保留在他们自己的 JAR文件,但是你可以包括这个EJB JAR文件在主服务 得一个存档(热)部署单元。请教JBoss AS 文档了解更多信息关于JMX服务和EJB 部署。

第四章。持久化类

表的内容

4.1. 一个简单的POJO的例子

以下4.4.1. 实现一个无参数的构造函数

4 1 2. 提供一个标识符属性

- 4.1.3. 喜欢不是final类(半可选)
- 4.1.4. 声明访问器和调整器为持久字段(可选)

- 4.2. 实现继承
- 4.3. 实施 equals() 和 hashCode()
- 4.4. 动态模型
- 4.5. Tuplizers
- 4.6. EntityNameResolvers

持久化类是类在一个应用程序,实现实体的业务问题 (例如。客户和订单在一个电子商务应用程序)。术语“持久化”这意味着类能够坚持,不是说他们是在持续状态(见吗 11.1节,“Hibernate对象状态”讨论)。

如果这些类冬眠最好遵循一些简单的规则,也被称为普通旧式Java对象(POJO)编程模型。然而,这些规则是硬要求。事实上,Hibernate假设非常小关于自然的持久对象。你可以表达一个域模型在其他方面(使用的树木 java util地图 实例,例如)。

4.1. 一个简单的POJO的例子

例4.1. 简单的POJO代表一只猫

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

public class Cat {
    private Long id; // identifier

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

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

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

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

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

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

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

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

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }

    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
}
```

```

public Set getKittens() {
    return kittens;
}

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

```

四个主要的规则是持久化类的文章中更详细地探讨以下部分。

以下4.4.1。实现一个无参数的构造函数

猫 有一个无参数的构造函数。所有的持久化类必须有一个默认的构造函数(可以非公有制),Hibernate可以实例化它们使用java的构造函数newInstance()。建议这个构造函数是定义在至少包能可见度为了运行时代理正常工作。

4 1 2。提供一个标识符属性

注意

历史上这是考虑选项。同时还(还)没有执行,应该考虑这一点。一个弃用功能,它将完全要求提供一个标识符属性在一个即将发布。

猫 拥有一个属性命名 id。这个属性映射到主键列(s)的底层数据库表。标识符的类型属性可以是任何“基本”类型(见吗??)。看到 9.4节,“组件作为复合标识符”关于映射组合(多列)标识符。

注意

标识符不一定需要识别列(s)在数据库中物理定义作为一个主键。他们应该只是识别列,可以用来唯一地标识行在底层表。

我们建议您申报的标识符属性始终命名持久化类,你使用一个可以为空(即。非基本的)类型。

4 1 3。喜欢不是final类(半可选)

Hibernate的中心要素,代理(延迟加载),取决于要么不是final的持久化类,或实现一个接口,声明所有的公共方法。你可以坚持最后类实现一个接口,不与Hibernate;你不会,但是,可以使用代理将延时关联获取最终限制你的选择性能调优。坚持一个最后类,它没有实现一个“完整”的接口必须禁用代理生成。看到 例4.2,“禁用代理 hbm xml”和 例4.3,“禁用代理在注释”。

例4.2。禁用代理在 hbm xml

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

例4.3。禁用代理在注释

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

如果最后类并实现一个合适的接口,你可以另外告诉Hibernate使用接口而不是在生成代理。看到 例4.4,“代理一个接口 hbm xml”和 例4.5,“代理接口在注释”。

例4.4。代理接口在 hbm xml

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

例4.5. 代理接口的注释

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

你还应该避免声明 `公众最终` 方法,因为这将再次限制 `能够生成代理` 从这个类。如果你想使用一个类 `公众最终` 方法,您必须显式禁用代理。再次,看到 例4.2,“禁用代理 `hbm.xml`” 和 例4.3,“禁用代理在注释”。

4.1.4. 声明访问器和调整器为持久字段(可选)

猫 `声明访问器` 方法为所有持久字段。其他许多ORM 工具直接持续实例变量。最好是提供一个间接层之间的关系 模式和类的内部数据结构。默认情况下,Hibernate持续javabeans样式 属性和识别方法名称的形式 `getFoo`, `isFoo` 和 `setFoo`。如果需要,您可以切换到直接字段访问特定 属性。

属性需要 `不被声明为公共的`。Hibernate可以坚持一个属性声明与 `包`, `保护` 或 `私人` 能见度为好。

4.2. 实现继承

一个子类也必须遵守第一和第二规则。它继承了 它的标识符属性从超类中,猫。对于 示例:

```
package eg;

public class DomesticCat extends Cat {
    private String name;

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

4.3. 实施 equals() 和 hashCode()

你必须重写 `equals()` 和 `hashCode()` 如果你的方法:

- ▶ 打算把持久化类的实例在一个 集 (推荐的方式来表示多值 协会); 和
- ▶ 打算使用回贴的分离实例

Hibernate保证持续的身份(数据库的等价性 行)和Java身份只在一个特定的会话范围。当你 混合在不同会话实例检索,您必须实现 `equals()` 和 `hashCode()` 如果你希望 有意义的语义 集 年代。

最明显的方式是实现 `equals()` / `hashCode()` 通过比较 标识符值的两个对象。如果该值是相同的,都必须 相同的数据库行,因为他们平等的。如果两个被添加到一个 集 ,您将只有一个元素 集)。不幸的是,您不能使用该 方法 生成的标识符。Hibernate只会分配标识符值 对象持久性;一个新创建的实例将没有任何 标识符值。此外,如果一个实例是,当前未保存 一个集,保存它将分配一个标识符值 对象。如果 `equals()` 和 `hashCode()` 基于标识符值。散列码会改变,打破 合同的 集。看到 Hibernate网站 一个完整的讨论这个问题。这不是一个Hibernate问题,但是 正常的Java语义对象的身份与平等。

建议您实现 `equals()` 和 `hashCode()` 使用 业务关键 平等。业务关键平等意味着 `equals()` 方法比较只有属性,表格 业务关键。这是一个关键,明确我们的实例在现实 世界(一个 自然 候选关键字):

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

        final Cat cat = (Cat) other;

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

        return true;
    }

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

```
}  
}
```

一个业务键不需要作为数据库主要 关键候选人(见 部分13.1.3节,“考虑对象的身份”)。 不变的或独特的属性通常适合业务 关键。

4.4. 动态模型

注意

以下特性目前正在考虑 实验和在不久的将来可能会改变。

持久化实体不一定必须表示为 POJO类或JavaBean对象在运行时。 Hibernate也支持 动态模型(使用 地图 年代的 地图 年代在运行时)。 通过这种方法,你不写持久化类,只有映射文件。

默认情况下,工作在正常模式下冬眠POJO。 你可以设置一个 默认的实体表示为一个特定的模式 SessionFactory 使用默认实体模式 配置选项(见 表格3.3,“Hibernate配置属性”)。

下面的例子演示了表示使用 地图 年代。 首先,在映射文件的一个 实体名称 必须声明相反的,或在吗 除了,类名:

```
<hibernate-mapping>  
  <class entity-name="Customer">  
    <id name="id"  
      type="long"  
      column="ID">  
      <generator class="sequence"/>  
    </id>  
    <property name="name"  
      column="NAME"  
      type="string"/>  
    <property name="address"  
      column="ADDRESS"  
      type="string"/>  
    <many-to-one name="organization"  
      column="ORGANIZATION_ID"  
      class="Organization"/>  
    <bag name="orders"  
      inverse="true"  
      lazy="false"  
      cascade="all">  
      <key column="CUSTOMER_ID"/>  
      <one-to-many class="Order"/>  
    </bag>  
  </class>  
</hibernate-mapping>
```

虽然协会声明使用目标类的名字, 目标类型的协会也可以是一个动态的实体,而不是一个 POJO。

在设置了默认的实体模式 动态地图 为 SessionFactory, 您可以在运行时使用 地图 年代的 地图 年代:

```
Session s = openSession();  
Transaction tx = s.beginTransaction();  
  
// Create a customer  
Map david = new HashMap();  
david.put("name", "David");  
  
// Create an organization  
Map foobar = new HashMap();  
foobar.put("name", "Foobar Inc.");  
  
// Link both  
david.put("organization", foobar);  
  
// Save both  
s.save("Customer", david);  
s.save("Organization", foobar);  
  
tx.commit();  
s.close();
```

的一个主要优势动态映射是快速周转 时间为原型,而不需要实体类的实现。 然而,你失去了编译时类型检查,可能会处理 许多在运行时异常。 结果的Hibernate映射, 数据库模式很容易被归一化和声音,允许添加一个 适当的域模型实现以后之上。

实体表示模式也可以设置在每一个 会话 基础:

```

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

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

```

请注意,在调用 getSession() 使用 一个 EntityMode 在 会话 API, 不是 SessionFactory 。 这样,新 会话 股票底层JDBC连接, 交易和其他上下文信息。 这意味着你不需要 叫 flush() 和 close() 在 二级 会话 ,还把事务和 连接处理,主要的工作单元。

Tuplizers 4.5.

org.hibernate.tuple.Tuplizer 和它的子接口负责 管理一个特定的表示一块数据鉴于表示的 org.hibernate.EntityMode 。 如果一个给定的块数据被认为是一个数据 结构,然后一个tuplizer就是知道如何创建这样一个数据结构,如何提取 值从这样一个 数据结构和如何将值放入这样一个数据结构。 例如,对于 POJO实体模式,相应的tuplizer知道如何通过其构造函数创建 POJO。 它还知道如何访问POJO属性使用定义的属性访问器。

有两种类型的Tuplizers(高级):

- » org.hibernate.tuple.entity.EntityTuplizer 这是 负责管理上述合同关于实体
- » org.hibernate.tuple.component.ComponentTuplizer 做 相同的组件

用户还可以塞在自己的tuplizers。 也许你需要 java util地图 实现其他比 java . util . hashmap 被使用在动态地图实体模式。 或也许你 需要定义一个不同的代理生成策略使用的默认情况下。 都将会实现 通过定义一个自定义tuplizer实现。 Tuplizer定义附加到实体或组件 他们是为了管理映射。 回到我们的例子 客户 实体, 例4.6, “指定自定义tuplizers在注释” 显示了如何指定一个自定义 org.hibernate.tuple.entity.EntityTuplizer 使用注释而 例4.7, “指定自定义tuplizers在 hbm xml ” 显示了如何做相同的 hbm xml

例4.6. 指定自定义tuplizers在注释

```

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

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

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

```

例4.7. 指定自定义tuplizers在 hbm xml

```

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

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

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

```

EntityNameResolvers 4.6.

org.hibernate.EntityNameResolver 是一个合同为解决实体的名字吗 一个给定的实体实例。 这个接口定义了一个方法 resolveEntityName 这是通过实体实例并将返回适当的实体名称(空是吗 允许和将表明,解析器不知道如何解决的实体名称指定的实体 实例)。 一般来说,一个 org.hibernate.EntityNameResolver 会 是最有用的情况动态模型。 一个例子是使用代理接口作为你的 域模型。 hibernate测试套件有一个这样的例子使用下确切的风格 org.hibernate.test.dynamicentity.tuplizer2 。 这里是一些代码从那包 为插图。

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

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

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

    public String getEntityName() {
        return entityName;
    }

    public HashMap getData() {
        return data;
    }
}

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

    // various other utility methods ....
}

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

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

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

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

```

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

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

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

```

为了注册一个 `org.hibernate.EntityNameResolver` 用户必须:

1. 实现一个自定义tuplizer(见 4.5节, “Tuplizers”),实现 这个 `getEntityNameResolvers` 方法
2. 把它登记到 `org.hibernate.impl.SessionFactoryImpl` (这是 实现类 `org.hibernate.sessionfactory`)使用 `registerEntityNameResolver` 法。

第五章。基本的O / R映射

表的内容

5.1. 映射声明

- 5.1.1. 实体
- 5.1.2中所述。标识符
- 5.1.3. 乐观锁定特性(可选)
- 5 1 4. 财产
- 是5.1.5. 嵌入对象(aka组件)
- 5—6. 继承策略
- 5—7. 映射一个对一个,另一个许多联想
- 5 1 8. 自然id
- 5 1 9. 任何
- 5—10. 属性
- 5 1 11. 一些hbm。xml特异性

5.2. Hibernate类型

- 5 2 1. 实体和值
- 5.2.2. 基本值类型
- 5 2 3. 自定义值类型

5.3. 映射一个类不止一次

- 5.4. SQL引用标识符
- 5.5. 生成的属性
- 5.6. 柱变形金刚:读和写表达式
- 5.7. 辅助数据库对象

5.1. 映射声明

对象/关系映射可以定义为三个 方法:

- » 使用Java 5注释(通过Java持久性2 注释)
- » 使用JPA 2 XML部署描述符(章所述 XXX)
- » 使用Hibernate遗留的XML文件的方法称为 hbm.xml

注释被分成了两类,逻辑映射 注释(描述对象模型之间的联系,两个 实体等)和物理映射注释(描述 物理模式、表、列、索引等)。我们将把注释 从两个类别在以下代码示例。

JPA注释的 `javax.persistence.*` 包。Hibernate特定扩展都在 `org.hibernate`注释。*。你最喜欢的IDE可以 自动完成注释和它们的属性(即使没有一个 特定的“JPA” 插件,因为JPA注释是纯Java 5 注释)。

下面是一个示例的映射

```

package eg;

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

```



```

public class Cat {

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

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

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

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

    @NotNull @Column(updatable=false)
    public String getSex() { return sex; }
    public void setSex(String sex) { this.sex = sex; }
    private String sex;

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

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

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

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

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

@Entity
public class Dog { ... }

```

遗留hbm。xml方法使用一个xml模式设计 可读性和手编。映射语言以java为中心的、意义,映射是围绕持久化类声明和不是表声明。

请注意,尽管许多Hibernate用户选择编写 XML的手,许多工具存在生成映射文件。 这些包括XDoclet,Middlegen和AndroMDA。

下面是一个示例映射:

```

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

<hibernate-mapping package="eg">

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

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

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

        <property name="weight"/>

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

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

        <property name="sex"

```

```

        not-null="true"
        update="false"/>

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

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

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

<subclass name="DomesticCat"
  discriminator-value="D">

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

</subclass>

</class>

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

</hibernate-mapping>

```

现在我们将讨论的概念映射文档(包括 注释和XML)。我们只会描述,然而,该文档 元素和属性在运行时使用Hibernate。映射文档还包含一些额外的可选的属性和元素 影响数据库模式导出的模式导出工具(用于 的例子, 过的非null的 属性)。

5.1.1. 实体

一个实体是一个普通Java对象(aka POJO)这将是 坚持以冬眠。

标记一个对象作为一个实体在注释,使用 `entity` 注释。

```

@Entity
public class Flight implements Serializable {
    Long id;

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

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

```

差不多就是这样,其余的是可选的。 但是任何 选项来调整你的实体映射,让我们探索它们。

`@table` 允许您定义表的实体 将持续到。 如果未定义,表名是不合格的 类名的实体。 你也可以选择定义目录, 模式以及独特的约束在桌子上。

```

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

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

```

约束名称是可选的(如果还未下定论生成)。 这个 列名构成约束对应列名称 之前定义的Hibernate namingStrategy 是 应用。

提示

一定要使用数据库级的列的名称 `columnNames` 财产 `@UniqueConstraint`。 例如,虽然对于简单类型的 数据库级列名称可能同实体级属性名称,这是经常的 不是这样的关系属性。

`@Entity.name` 允许您定义的快捷方式名称 的实体可以用于以jp - ql和HQL查询。 它默认 不合格的类的类名。

Hibernate超越JPA规范和提供额外的 配置。 他们中的一些是托管在 `@org.hibernate.annotations.Entity` :

- ▶ `dynamicInsert / dynamicUpdate` (默认值为false):指定 插入 / 更新 SQL应该 在运行时生成和只包含列的值 非空。 这

个 动态更新 和 动态插入 设置不继承了 子类。 虽然这些设置可以提高性能在一些 情况下,他们实际上会降低性能在其他 人。

- ▶ selectBeforeUpdate (默认为false): 指定Hibernate应该 从来没有 执行 一个SQL 更新 除非,这是肯定的一个对象 实际上是修改。 只有当一个临时对象 关联到一个新的会话使用 Update(), 将Hibernate执行一个额外的SQL 选择 到 确定 一个 更新 实际上是要求。 使用的 - before - update 通常会减少 性能。 它有助于防止数据库更新触发器被 如果你重 叫不必要的图分离实例 一个 会话。
- ▶ 多态性 (默认为 隐式):确定隐式或 使用显式查询多态性。 隐式 多态性意味着类的实例将被返回 一个查询,名称或实施的 任何超类接口或类, 实例的任何子类的类将被返回 一个查询,名称类本身。 明确 多态性意味着类实例将返回只有 明确 的名字,查询类。 查询的类名称 将只返回子类的实例映射。 对于大多数的目的, 默认 多态性=隐式 是 适当的。 明确的多态 性是有用,当两个不同的 类映射到相同的表这允许一个 “轻量级” 类,该类包含表列的一个子集。
- ▶ 持续程序 :指定一个自定义 ClassPersister 。 这个 持续程序 属性允许您定制的持久性策略用于 类。 你可以,例如,指定自己 的子类 org.hibernate.persister.EntityPersister ,或者你 甚至可以提供一个完全的新实现接口 org.hibernate.persister.ClassPersister 这 实现,例如,坚持通过存储过程调用, 序列化到平面文件或LDAP。 看到 org.hibernate.test.CustomPersister 对于一个简单的 的例子, “持久化” 一个 Hashtable 。
- ▶ optimisticLock (默认为 版本):决定了乐观锁定 策略。 如果您启用 dynamicUpdate ,你会 可以选择乐观锁定策略:
 - 版本 :检查版本/时间戳 列
 - 所有 :检查所有列
 - 脏 :检查改变列, 允许一些并发更新
 - 没有 :不要使用乐观 锁定

这是 强烈 建议您使用 版本/时间戳列与Hibernate乐观锁定。 这个策略优化性能和正确处理 更改分离实例(即当 会话合并() 是使用)。

提示

一定要进口 @javax.persistence.Entity 一个类作为其标志 实体。 这是一个常见的错误导 入 @org.hibernate.annotations.Entity 通过 事故。

一些实体不是可变的。 他们不能被更新 通过应用程序。 这允许Hibernate做一些轻微的性能 优化。 使用 @ immutable 注 释。

你也可以改变Hibernate处理延迟初始化 对于这类。 在 @Proxy ,使用 懒惰 = false禁用惰性抓取(不是 推荐)。 您还可以指 定一个接口用于懒惰 初始化代理(默认为类本身):使用 proxyClass 在 @Proxy 。 Hibernate最初将返回代理(Javassist和 CGLIB) 实现指定的接口。 持久对象加载当 方法调用的代理。 看到 “初始化集合和 代理” 下面。

@BatchSize 指定一个 “批量大小” 获取这个类的实例的标识符。 没有加载实例 加载一次批量大小(默认为1)。

你可以具体的任意SQL,条件时使用 检索对象的类。 使用 @Where 对于 那。

同样, @Check 允许您定义一个 SQL表达式用来生成一个多行 检查 约束自动模式生成。

是没有什么区别的一个视图和基本表 Hibernate映射。 这是透明的在数据库级,虽然 一些数据库管理系统不支持的观点正确, 特别是与更新。 有时你想要使用一个视图,但是你不能创建一个 数据库(即遗留模式)。 在这种情况下,您可以绘制一个 不可变的 和只读实体以给定的SQL subselect表达式使用 @org.hibernate.annotations.Subselect :

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

声明表同步这个实体,确保 自动刷新发生正确,查询导出实体 不返回陈旧的数据。 这个 < subselect > 是 既有一个属性和一 个嵌套映射元素。

现在我们将探索相同的选项使用hbm。 XML结构。 你可以声明一个持久化类使用 类 元素。 例如:

```
<class
name="ClassName"
table="tableName"
discriminator-value="discriminator_value"
mutable="true|false"
schema="owner"
catalog="catalog"
proxy="ProxyInterface"
dynamic-update="true|false"
dynamic-insert="true|false"
select-before-update="true|false"
polymorphism="implicit|explicit"
where="arbitrary sql where condition"
persister="PersisterClass"
batch-size="N"
optimistic-lock="none|version|dirty|all"
lazy="true|false"
entity-name="EntityName"
check="arbitrary sql check condition"
```

```

rowxml:id="rowid"
subselect="SQL expression"
abstract="true|false"
node="element-name"
/>

```

-  名称 (可选):完全限定Java 类名的持久类或接口。 如果这个属性 是失踪,据推测映射是一个非pojo吗 实体。
-  表 (可选,默认了 不合格的类名):名称的数据库表。
-  discriminator值 (可选,默认 的类名):一个值,区分个体 子类,用于多态行为。 接受 价值观包括 空 和 不空 。
-  可变 (可选,默认 真正的):指定类的实例 有(无)可变的。
-  模式 (可选):覆盖模式 名指定的根 < hibernate映射> 元素。
-  目录 (可选):覆盖目录 名指定的根 < hibernate映射> 元素。
-  代理 (可选):指定一个接口 用于初始化代理懒惰。 你可以指定名称 类本身。
-  动态更新 (可选,默认 假):指定 更新 SQL应该在运行时生成的和 只能包含的那些列值已经改变了。
-  动态插入 (可选,默认 假):指定 插入 SQL应该在运行时生成的和 只包含列的值不为null。
-  - before - update (可选,默认 到 假):指定Hibernate应该 从来没有 执行一个SQL 更新 除非,这是肯定的一个对象 其实修改。 只有当一个临时对象 关联到一个新的会话使用 Update(), 将Hibernate执行一个额外的SQL 选择 到 确定一个 更新 实际上是 要求。
-  多态性 (可选,默认 隐式):确定隐式或 使用显式查询多态性。
-  在 (可选):指定一个任意的 SQL 在 当检索条件使用 这个类的对象。
-  持续程序 (可选):指定一个自定义的 ClassPersister 。
-  批量大小 (可选,默认 1):指定一个 “批量大小” 获取 这个类的实例的标识符。
-  乐观锁 (可选,默认 版本):决定了乐观锁定 策略。
- (16) 懒惰 (可选):惰性抓取可以 残疾人通过设置 懒= " false " 。
- (17) 实体名称 (可选,默认了 类名):Hibernate3允许一个类要映射的多个 次,可能不同的表。 它还允许实体 映射,为地图或 XML的Java级别。 在 这些情况下,您应该提供一个明确的任意名称 实体。 看到 4.4节, “动态模型” 和 吗? ?? 为更多的信息。
- (18) 检查 (可选):使用一个SQL表达式 来生成一个多行 检查 约束为 自动模式生成。
- (19) rowid (可选):Hibernate可以使用 ROWID数据库上。 在Oracle上,例如,Hibernate可以使用 rowid 额外的列快速 更新一旦 选项设置 rowid 。 一个是ROWID 实现细节和代表的物理位置 存储元组。
- (20) subselect (可选):地图一个不可变的 和只读实体数据库subselect。 这是有用的,如果 你想有一个视图,而不是一个基本表。 请看以下 更多的信息。
- (21) 文摘 (可选):用来标志 抽象超类在 <联盟子类> 层次结构。

它是可以接受的持久化类的命名是一个 接口。 你可以声明该接口的实现类使用 这个 <子类> 元素。 你可以保存任何 静态内部类。 指定类名使用 标准的形式即。 e g foo \$酒吧 。

下面是如何做一个虚拟视图(subselect)在XML:

```

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

```

这个 < subselect > 可既作为一个吗 属性和一个嵌套映射元素。

5.1.2中所述。 标识符

映射类 必须 宣布主键 数据库表的列。 大多数类还将有一个 javabean样式属性持有独特的标识符的一个 实例。

马克标识符属性 @ id 。

```

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

```






在hbm. xml,使用 < Id > 元素 定义了该属性映射到主键列。

```

<id
  name="propertyName"
  type="typename"
  column="column_name"
  unsaved-value="null|any|none|undefined|id_value"
  access="field|property|ClassName">

```

```
node="element-name|@attribute-name|element/@attribute|"  
<generator class="generatorClass"/>  
</id>
```

-  名称 (可选)的名字 标识符属性。
-  类型 (可选):一个名字表明 Hibernate类型。
-  列 (可选,默认了 属性名):主键列的名称。
-  未保存的价值 (可选,默认为一个 “明智” 值):一个标识符属性值表示的 实例是新实例化(未保存的),区分它 分离实例被保存或装载在前面的 会话。
-  访问 (可选,默认 财产):战略Hibernate应该使用 访问属性值。

如果 名称 属性缺失,这是假定 这类没有标识符属性。

这个 未保存的价值 属性几乎从来没有 需要在Hibernate3的确没有相应元素的 注释。

你也可以声明标识符作为一个复合的标识符。 这允许访问遗留数据与组合键。 它的使用是 强烈劝阻其他东西。

5 1 2 1. 复合标识符

您可以定义一个复合主键通过几次 语法:

- » 使用一个组件类型来表示标识符,并将其映射 作为一个属性的实体:你然后带注释的属性作为 @EmbeddedId 。 组件类型必须 可序列化的 。
- » 映射多个属性作为 @ id 属性:标识符类型是那么实体类本身 和需要 可序列化的 。 这种方法 不幸的是没有标准,只有支持 吗 Hibernate。
- » 映射多个属性作为 @ id 属性和声明一个外部类的标识符 类型。 这类,它需要 可序列化的 ,是在实体上宣布通过 这个 @IdClass 注释。 标识符 类型必须包含相同的属性标识符属性 实体:每个属性的名字必须相同,其类型必须 如果是相同的 实体属性的基本类型,它的 类型必须是主键的类型关联的实体 如果实体属性是一个协会(要么 @OneToOne 或 @ManyToOne)。

正如您可以看到的最后情况远非明显。 它已经被 继承了黑暗时代的EJB 2向后兼容性和 我们建议你不要使用它(为简单起见)。

让我们探索所有三个案例中使用的例子。

5 1 2 1 1. id属性使用一个组件类型

这是一个简单的例子 @EmbeddedId 。

```
@Entity  
class User {  
    @EmbeddedId  
    @AttributeOverride(name="firstName", column=@Column(name="fld_firstname"))  
    UserId id;  
  
    Integer age;  
}  
  
@Embeddable  
class UserId implements Serializable {  
    String firstName;  
    String lastName;  
}
```

你可以注意到 userId 类是 可序列化的。 覆盖列映射,使用 @AttributeOverride 。

嵌入式id可以本身含有的主键 相关的实体。

```
@Entity  
class Customer {  
    @EmbeddedId CustomerId id;  
    boolean preferredCustomer;  
  
    @MapsId("userId")  
    @JoinColumns({  
        @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),  
        @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")  
    })  
    @OneToOne User user;  
}  
  
@Embeddable  
class CustomerId implements Serializable {  
    UserId userId;  
    String customerNumber;  
  
    //implements equals and hashCode  
}
```

```

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

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

    //implements equals and hashCode
}

```

在嵌入式id对象,该协会表示为 标识符关联的实体。 但是你可以连接它的值 定期的协会在实体通过 @MapsId 注释。 这个 @MapsId 值对应的属性名称 嵌入对象的id包含相关的实体的 标识符。 在数据库中,这意味着 客户用户 和 customerid userid 属性共享相同的 底层柱(用户第 在这种情况下)。

提示

组件类型用作标识符必须实现 equals() 和 hashCode() 。

在实践中,您的代码只设置 客户用户 属性和用户id值 复制到。 冬眠 customerid userid 财产。

警告

id值可以被复制到冲洗时间,不要相信 在这之前冲洗时间。

虽然不支持在JPA,Hibernate可以让你把你的 协会直接在嵌入式id组件(而不是拥有 使用 @MapsId 注释)。

```

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

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

    //implements equals and hashCode
}

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

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

    //implements equals and hashCode
}

```

现在让我们重写这些例子使用hbm xml 语法。

```

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

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

```

首先一个简单的例子:

```

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

```

然后一个例子展示了一个协会可以映射。

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

  <property name="preferredCustomer"/>

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

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

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

注意几件事在前面的例子:

- ▶ 订单属性(列)问题。它必须是相同的在协会和主键的 相关联的实体
- ▶ 多数人使用相同的列作为主键 因而必须标记为只读 (可插入的 和 可更新的 为false)。
- ▶ 与 @MapsId ,id值 关联的实体是不透明的复制,检查 外国 id发生器更多 信息。

最后一个例子展示了如何直接在地图协会 嵌入式id组件。

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

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

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

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

这是推荐的方法复合标识符映射。以下选项不应该被考虑,除非一些 约束存在。

5 1 2 1 2. 多个id属性没有标识符类型

另一个,更自然的方法是把 @ id 在多个属性的实体。这种方法仅支持通过Hibernate(不是JPA兼容)但是 不需要额外的可嵌入的组件。

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

  @Id String customerNumber;

  boolean preferredCustomer;

  //implements equals and hashCode
}

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

@Embeddable
class UserId implements Serializable {
```

```
String firstName;
String lastName;

//implements equals and hashCode
}
```

在这种情况下 客户 是自己的 标识符表示:它必须实现 可序列化的 和必须实现 equals() 和 hashCode() 。

在hbm. xml,相同的映射:

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

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

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

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

5 1 2 1 3. 多个id属性与一个专用的标识符 类型

@IdClass 在一个实体指向 类(组件)代表了标识符的类。 这个 属性标记 @ id 在实体上必须有 他们相应的产权 @IdClass 。 返回类型的搜索双属性必须是相同的 基本属性或必须对应标识符类的 相关联的实体的关系。

警告

这种方法是继承了EJB 2天,我们 不推荐使用它。 但是,这毕竟是你的应用程序 和Hibernate 支持它。

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

  @Id String customerNumber;

  boolean preferredCustomer;
}

class CustomerId implements Serializable {
  UserId user;
  String customerNumber;

  //implements equals and hashCode
}

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

  //implements equals and hashCode
}

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

  //implements equals and hashCode
}
```

客户 和 customerId 有相同的属性呢 customerNumber 以及 用户 。 customerId 必须 可序列化的 和实施 equals() 和 hashCode() 。

虽然不是JPA标准,Hibernate咱们你声明 香草相关属性 @IdClass 。

```
@Entity
```



```

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

    @Id String customerNumber;

    boolean preferredCustomer;
}

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

    //implements equals and hashCode
}

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

    //implements equals and hashCode
}

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

```

这个功能是有限的利益,尽管你可能已经选择了 @IdClass 方法保持 JPA兼容或者你有一个非常扭曲的心灵。

这里有相当于hbm上。xml文件:

```

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

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

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

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

```

5 1 2 2. 标识符生成器

Hibernate可以生成并填充为你标识符值自动。这是推荐的方法在“业务”或“自然”id(尤其是复合id)。

Hibernate提供了各种生成策略,让我们探索 最常见的第一,恰好是标准化的 JPA:

- » 身份:支持身份列在DB2、MySQL、MS SQL 服务器、Sybase和HypersonicSQL。返回标识符的类型 长,短 或 int。
- » 序列(称为 seqhilo 在Hibernate): 使用一个嗨/ lo算法高效地生成标识符的类型 长,短 或 int ,给一个指定的数据库序列。
- » 表(称为 MultipleHiLoPerTableGenerator 在Hibernate) :使用一个嗨/ lo算法高效地生成的标识符 类型 长,短 或 int ,给定一个表和列的一个来源的嗨值。 hi / lo算法生成标识符,是独一无二的 只有一个特定的数据库。
- » 汽车:选择 身份,序列 或 表 根据 在底层数据库的功能。

重要

我们建议所有新项目使用这些新的增强的 标识符生成器。 他们是默认关闭的实体 使用注释,但可以激活使用 hibernate id新发电机映射= true 。 这些新 发电机更高效、更接近JPA 2规范 语义。

然而他们不向后兼容现有的 Hibernate应用程序(如果一个序列为基础或表用于id 一代)。看到XXXXXXX 吗? ? 对于 更多的信息关于如何激活它们。

标记一个id属性生成,使用 @GeneratedValue 注释。 你可以指定 策略使用(默认 汽车)通过设置 策略。

```

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

@Entity
public class Invoice {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    Integer getId() { ... };
}

```

序列和表需要额外的配置,你可以设置使用 generator 和 @TableGenerator :

- » 名称 :名字的发电机
- » 表 / sequenceName : 表名或序列(违约的分别 hibernate序列 和 hibernate序列)
- » 目录 / 模式 :
- » initialValue :价值的id 是开始生成
- » allocationSize :增加数量 通过在分配id数字发生器

此外,表策略也让你定制:

- » pkColumnName :列名包含 实体标识符
- » valueColumnName :列名 包含标识符值
- » pkColumnValue :实体 标识符
- » uniqueConstraints :任何潜在的列 约束的表包含id

链接一个表或序列发生器与一个实际的定义 生成的属性,使用相同的名称的定义 名称 和发电机价值 发电机 如下所示。

```

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

```

发电机的范围定义应用程序或 类。 类定义的发电机是不可见的以外的其他类 和可以覆盖应用程序级别的发电机。 应用水平发电机被定义在JPA的XML部署描述符(见XXXXXX 吗? ??):

```

<table-generator name="EMP_GEN"
    table="GENERATOR_TABLE"
    pk-column-name="key"
    value-column-name="hi"
    pk-column-value="EMP"
    allocation-size="20"/>

//and the annotation equivalent

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

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

//and the annotation equivalent

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

```

如果一个JPA XML描述符(如 meta - inf / xml文件)用来定义 发电机, emp创 和 seq创 是应用程序级别的发电机。

注意

包级别定义不支持JPA 规范。 然而,你可以使用 @GenericGenerator 在包级别(参见 吗? ? ?)。

这些都是这四个标准JPA发电机。 Hibernate会 除此之外,提供额外的发电机或额外的选项 我们将在下面看到。 您也可以编写自己的自定义标识符 发生器通过实现 org.hibernate.id.IdentifierGenerator 。

定义一个自定义的生成器,使用 @GeneratedValue 注释(和它的复数 计数器部分 @GenericGenerators),描述 的类标识符生成器或其捷径名称(作为 下面描述)和一系列键/值参数。 当使用 @GeneratedValue 和分配它通过 @GeneratedValue.generator , @GeneratedValue.strategy 被忽略了:离开它吗 空白。

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

@Id @GeneratedValue(generator="trigger-generated")
@GenericGenerator(
    name="trigger-generated",
    strategy = "select",
    parameters = @Parameter(name="key", value = "socialSecurityNumber")
)
public String getId() {
```

恒的。xml方法使用可选的 <生成器> 子元素内 < Id > 。 如果需要任何参数 配置或初始化生成器实例,它们是通过使用 这个 < param > 元素。

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

5.1.2.2.1. 各种额外的发电机

所有的发电机实现接口 org.hibernate.id.IdentifierGenerator 。 这是一个 非常简单的接口。 一些应用程序可以选择为他们的 自己的专业实现,然而,Hibernate提供了一个范围 内置的实现。 快捷方式的名称为内置的 发电机如下:

增量

生成标识符的类型 长,短 或 int 这是 只有当没有其他过程的独特是将数据插入到 相同的表。 不要使用在吗 集群。

身份

支持身份列在DB2、MySQL、MS SQL 服务器、Sybase和HypersonicSQL。 返回的标识符是 类型的 长,短 或 int 。

序列

使用一个序列在DB2中,PostgreSQL,甲骨文,SAP DB, McKoi或发电机在包括。 返回的标识符 的类型是 长,短 或 int

希洛的

使用一个嗨/ lo算法高效地生成 标识符的类型 长,短 或 int ,给定一个 表和列(默认情况下 hibernate独特的关键 和下一个嗨 作为一个来源的分别)嗨 值。 hi / lo算法生成标识符,独特的只存在一个特定的数据库。

seqhilo

使用一个嗨/ lo算法高效地生成 标识符的类型 长,短 或 int ,给定一个 指定的数据库序列。

uuid

生成一个128位的UUID基于自定义算法。 生成的值被表示为一个字符串的32 hexadecimal位数。 用户也可以将其配置为使用 分离器(配置参数 "分离器")所分割的 hexadecimal位数为8 { 9 } 8 { 9 } 4 { 9 } 8 { 9 } 4。 注意 特别是,这是不同的比IETF RFC 4122 表示8-4-4-4-12。 如果你需要RFC 4122兼容 uuid,考虑使用 "uuid2" 生成器讨论 下面。

uuid2

生成一个IETF RFC 4122兼容(变种2) 128位的UUID。 确切的 "版本" (RFC术语)生成的 取决于使用可插拔的 "一代战略" (见 下面)。 能够生成值 java util uuid ,以 或作为一个字节数组 长度16(byte[16])。 "一代 战略" 被定义为接口 org.hibernate.id.UUIDGenerationStrategy 。 发电机定义2配置参数 定义生成策略的使用:

uuid创战略类

UUIDGenerationStrategy类的名字 使用

uuid创战略

UUIDGenerationStrategy命名的实例 使用

出箱,有以下策略:

- » org.hibernate.id.uuid.StandardRandomStrategy (默认)-生成"版本3" (又称, "随机") UUID值通过 randomUUID 方法 java util uuid
- » org.hibernate.id.uuid.CustomVersionOneStrategy ——生成"版本1 "UUID值,使用IP地址 因为mac地址不可用。 如果你需要mac 地址使用,可以考虑利用一个的 现有的第三方UUID发电机,嗅出 mac地址和整合它通过 org.hibernate.id.UUIDGenerationStrategy 合同。 这两个库的已知 写作包括 <http://johannburkard.de/software/uuid/> 和 <http://commons.apache.org/sandbox/id/uuid.html>

guid

使用一个数据库生成的SQL服务器上GUID字符串女士 和MySQL。

原生

选择 身份,序列 或 希洛的 根据功能的基础 数据库。

分配

让应用程序分配一个标识符 对象之前 save() 被称为。 这是 默认策略如果没有 <发生器> 元素是 指定的。

选择

检索一个主键,由数据库 触发器,通过选择行通过一些独特的关键和 获取主键值。

外国

使用标识符相关联的另一个对象。 它 通常用于联同一个吗 <一对一> 主键 协会。

序列的身份

一个专业序列生成策略 利用数据库序列的实际价值 代,但将其与JDBC3 getGeneratedKeys来 返回生成的标识符值作为 插入 语句的执行。 这个策略只支持 Oracle 10 g司机针对JDK 1.4。 评论这些 insert语句被禁用由于一个错误在甲骨文 司机。

5 1 2 2 2. 你好/ lo算法

这个 希洛的 和 seqhilo 发电机提供两个交替的实现嗨/瞧 算法。 第一个实现需要一个 “特殊” 的数据库 表来保存下一个可 用的 “嗨” 价值。 在支持着, 第二个使用一个oracle型序列。

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

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

不幸的是,您不能使用 希洛的 当 提供自己的 连接 冬眠。 当 Hibernate使用应用服务器数据源获取 连接应加入JTA,您必须 配置 manager_lookup_class 。

5 1 2 2 3. UUID算法

的UUID包含:IP地址,JVM的启动时间, 精确到一个季度第二,系统时间和计数器的值 在JVM中是独一无二的。 它是不可能获得 一个MAC地址 从Java代码或内存地址,所以这是最好的选择没有 使用JNI。

5 1 2 2 4. 标识列和序列

对于数据库,支持身份列(DB2、MySQL、 Sybase,MS SQL),您可以使用 身份 关键 一代。 对于数据库,支持序列(DB2、 Oracle、 PostgreSQL,包括SAP DB),McKoi,你可以使用 序列 风格键生成。 这两种 策略需要两个SQL查询来插入一个新的 对象。 对于 示例:

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

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

对于跨平台开发, 原生 战略将,这取决于能力的基础 数据库,选择从 身份, 序列 和 希洛的 策略。

5 1 2 2 5. 指定标识符

如果您希望应用程序指定的标识符,而不是 让Hibernate生成它们,您可以使用 分配 发电机。 这种特殊的发电机使用 的标识符 值已经分配给对象的标识符 财产。 发电机是用于当主键是一个自然的 关键不是一个代理键。 这是默认的行为如果你 没有指定 @GeneratedValue 也 <发生器> 元素。

这个 分配 发生器使Hibernate使用 未保存的价值= “定义” 。 这迫使Hibernate 去到数据库,以确定一个实例是短暂的或 超 然的,除非有一个版本或时间戳属性,或者你 定义 Interceptor.isUnsaved() 。

5 1 2 2 6. 主键指定触发器

Hibernate不生成DDL和触发器。它是只遗留模式。

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

在上面的示例中,有一个独特的价值属性命名 SOCIALSECURITYNUMBER。这是定义的类,作为一个自然键和一个代理键命名人身份证,其值是生成的触发。

5.1.2.2.7. 身份复印件(外国发生器)

最后,你可以问Hibernate来复制标识符另一个相关的实体。在Hibernate术语,它被称为一个外国发生器但是JPA映射读取更好,是鼓励。

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

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

或者

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

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

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

在hbm.xml使用下列方法:

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

5.1.2.3. 增强标识符生成器

3.2.3开始释放,有2个新的发电机这代表了两种不同的思考方面的标识符一代。第一个方面是数据库的可移植性;第二个是优化优化意味着你不需要查询数据库为每个请求为一个新的标识符值。这两个新发电机是为了代替一些命名发电机上面所述,在.x开始。然而,他们是在当前版本中,可以引用FQN。

第一个新发电机 org.hibernate.id.enhanced.SequenceStyleGenerator 这是有意的,首先,作为替代吗序列发电机和,其次,作为一个更好的可移植性发生器比原生。这是因为原生一般选择之间身份和序列哪有很大程度上不同的语义,可能导致微妙的问题应用程序可移植性盯上。 org.hibernate.id.enhanced.SequenceStyleGenerator,然而,在一个不同的方式取得可移植性。它选择 一个表或一个序列之间的数据库来存储递增值,根据功能的方言被使用。这之间的差异和原生是,表格和基于序列的存储有相同的语义。事实上,序列正是Hibernate试图仿真以其基于发电机。这个发电机有一个号码配置参数:

- » 序列名称 (可选,默认 hibernate序列):序列的名称 或表使用。
- » 初始值 (可选,默认 1):初始值来检索自 序列/表。在序列创建条件,这是类似于条款通常命名为“始于”。
- » increment_size (可选,默认 1):值,随后调用 序列/表应该有所不同。在序列创建术语,这是类似于条款通常命名为“增量通过”。
- » 力表使用 (可选,默认 假):我们应该强制使用一个表 支持结构尽管方言可能支持 序列?
- » 值列 (可选,默认 下瓦尔):只有相关的表 结构,它的名字是列在表这是 用来保存价值。
- » 喜欢序列每个实体 (可选的一 默认为 假):我们应该创建 单独的序列对于每个实体共享电流发生器 基于它的名字吗?
- » 序列每个实体后缀 (可选的一 默认为 seq):后缀添加到名称 专用的序列。
- » 优化器 (可选,默认 没有):看 部分5.1.2.3.1,“标识符生成器优化”

第二种新发电机 org.hibernate.id.enhanced.TableGenerator ,这是 打算,首先,作为一个替代 表 发电机,尽管它实际上功能更像 org.hibernate.id.MultipleHiLoPerTableGenerator ,和 其次,作为一个的重新实现 org.hibernate.id.MultipleHiLoPerTableGenerator 这 利用可插入优化的概念。 本质上这 发电机定义一个表能够持有许多不同的 增量值同时使用多个明显的键 行。 这个发电机有许多配置参数:

- ▶ table_NAME (可选,默认 hibernate序列):表名 被使用。
- ▶ 值列名称 (可选,默认 到 下瓦尔):列的名称的 表,用于保存价值。
- ▶ 段列名 (可选的一 默认为 序列名称)的名字 列在表,用来保存“段关键”。 这是价值,能识别哪些增量值 使用。
- ▶ 段值 (可选,默认 默认):“段关键” 的值 段我们想拉增量值为这个 发电机。
- ▶ 段值长度 (可选的一 默认为 255年):用于模式生成; 列的大小创建这段关键列。
- ▶ 初始值 (可选,默认 1):初始值来检索自 这个表。
- ▶ increment_size (可选,默认 1):值,随后调用 这个表应该有所不同。
- ▶ 优化器 (可选,默认 ??):看 部分5 1 2 3 1,“标识符生成器优化” 。

5 1 2 3 1. 标识符生成器优化

为标识符生成器,在数据库中存储值,这是低效的,他们袭击了数据库在每个 调用生成一个新的标识符值。 相反,你可以组织一个群他们在内存中,只有当你有了数据库 你的内存值组筋疲力尽。 这是所扮演的角色 可插拔的优化。 目前只有两个增强的发电机 (部分5 1 2 3,“强化标识符生成器” 支持这个 操作。

- ▶ 没有 (通常这是默认的结果 没有优化器指定);这将不会执行任何 优化和触及数据库,为每个 请求。
- ▶ 希洛的 :应用一个嗨/ lo算法约 数据库检索到的值。 从数据库的值为 这个优化器预计将顺序。 值 从数据库检索结构优化指出了“组数”。 这个 increment_size 是乘以该值在吗 内存来定义一组“嗨价值” 。
- ▶ 汇集 :与案件的 希洛的 ,这个优化器试图最小化 数量的支安打到数据库。 不过,在这里,我们只是存储 的起始值“下一组” 到数据库中 结构而不是一个序列值结合 内存中的分组算法。 在这里, increment_size 指的是价值观来 从数据库。

5 1 2 4. 部分标识符生成

Hibernate支持自动生成一些 标识符属性。 仅仅使用 @GeneratedValue 注释一个或几个id 属性。

警告

Hibernate团队一直觉得这种结构作为 从根本上错误的。 努力解决您的数据模型在使用 这个特性。

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

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

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

你也可以生成属性内 @EmbeddedId 类。

5.1.3. 乐观锁定特性(可选)

当使用长事务或对话,跨几个 数据库事务,它是有用的数据,以保证存储版本控制 ,如果相同的实体是更新两个对话,最后一个 提交修改将被告知,而不是覆盖其他 对话的作品。 它保证一些隔离,同时仍然允许 良好的可伸缩性和作品在阅读往往特别好 写有时情况。

您可以使用两种方法:一个专用版本号或一个 时间戳。

一个版本或时间戳属性不应该为空 分离实例。 Hibernate会发现任何实例使用一个空值 版本或时间戳作为过渡,不论什么其他 未保存的价值 指定的策略。 声明一个nullable版本或时间戳属性是一个简单的 为了避免问题与及动词在Hibernate回贴。

这是 特别适合人们使用指定的标识符或复合 键 。

5 1 3 1. 版本号

您可以添加乐观锁定能力,一个实体使用 version 注释:

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

这个版本的财产将被映射到 OPTLOCK 列,和实体管理器将使用它 来检测冲突的更新(防止丢失更新您可能 否则看到的最后提交的获胜策略)。








版本字段可以是一个数值。 Hibernate支持任何 如果你的类型定义和实施适当的 UserVersionType 。

应用程序必须不改变版本号建立的 Hibernate在任何方式。 人为地增加版本号, 检查在Hibernate实体管理器的参考文档 LockModeType.OPTIMISTIC_FORCE_INCREMENT 或 LockModeType.PESSIMISTIC_FORCE_INCREMENT 。

如果版本号是由数据库生成(通过 [@org.hibernate.annotations.Generated\(GenerationTime.ALWAYS\)](#) 引发例如),确保使用

声明一个版本属性在hbm. xml,使用:

```
<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 (可选,默认了 属性名):列的名称控股版本 号码。
-  名称 :属性的名称的 持久化类。
-  类型 (可选,默认 整数):类型的版本 号码。
-  访问 (可选,默认 财产):策略使用Hibernate 访问属性值。
-  未保存的价值 (可选,默认 未定义):一个版本的属性值 表明新实例化一个实例(未保存的), 区分它从分离实例保存或 装载在前一交易日。 未定义 指定的标识符属性值应该是 使用。
-  生成 (可选,默认 从来没有):指定该版本属性 值是由数据库生成。 看到讨论 [生成的属性](#) 更多 信息。
-  插入 (可选,默认 真正的):指定是否版本列 应包括在SQL insert语句。 它可以设置为 假 如果数据库列定义 一个默认值 0 。

5 1 3 2. 时间戳

或者,您可以使用一个时间戳。 时间戳是少 安全实现的乐观锁定。 然而,有时一个 应用程序可能使用时间戳的方式。

简单的一个属性类型。 马克 日期 或 日历 作为 version 。

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







当使用时间戳版本你可以告诉Hibernate在哪里 获取时间戳值- 数据库或JVM——可选 添加 [@org.hibernate.annotations.Source](#) 注释的属性。 可能值的值属性 注释是 [org.hibernate.annotations.SourceType.VM](#) 和 [org.hibernate.annotations.SourceType.DB](#) 。 这个 默认是 [SourceType.DB](#) 过去也曾在吗 案件没有 [@Source](#) 注释在 所有。

就像对于版本号,时间戳也可以 生成的数据库而不是冬眠。 为此,可以使用 [@org.hibernate.annotations.Generated\(GenerationTime.ALWAYS\)](#)。

在hbm. xml,使用 < timestamp > 元素:

```
<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 (可选,默认了 属性名):名称列着 时间戳。
-  名称 :一个javabeen样式的名称 属性的Java类型 日期 或 时间戳 持久化类的。
-  访问 (可选,默认 财产):战略Hibernate使用 访问属性值。
-  未保存的价值 (可选,默认 空):一个版本的属性值 表明新实例化一个实例(未保存的), 区分它从分离实例保存或 装载在前一交易日。 未定义 指定的标识符属性值应该是 使用。
-  源 (可选,默认 VM):应该Hibernate检索 时间戳值吗? 从数据库中,或从当前 JVM ? 基于时间戳的开销,因为产生 Hibernate必须触及数据库为了确定 “下一个 价值” 。 更安全的做法是在集群环境中使用。 并不是所有的 方言 已知支持检索 数据库当前的时间戳。 其他人可能也会不安全 使用在锁定由于缺乏精密(Oracle 8,因为 例)。
-  生成 (可选,默认 从来没有):指定该时间戳 属性值实际上是由数据库生成。 看到 讨论 [生成 属性](#) 为更多的信息。

注意

< timestamp > 相当于 <版本类型= “时间戳” > 。 和 <时间戳源= “分贝” > 相当于 <版本类型= “dbtimestamp” >

5 1 4. 财产

你需要决定哪些属性需要存储 给定的实体。 这个注释驱动之间稍有不同 元数据和hbm. xml文件。

5 1 4 1. 属性映射与注释

在注释的世界,每一个非静态非瞬态 属性(字段或方法根据不同的访问类型)的一个实体 被认为是持久的,除非你将其标注为 @Transient 。 没有一个注释为你 财产相当于适当的 @Basic 注释。

这个 @Basic 注释允许您声明 对于一个属性的抓取策略。 如果设置为 懒惰 ,指定该属性应该是 当实例变量获取懒洋洋地在第一次访问。 它 需要构建时字节码插装,如果你的类是不 仪表化,属性级延迟加载是悄悄地忽略。 这个 默认是 急切 。 你也可以马克财产 没有可选的感谢 @Basic.optional 属性。 这将确保底层柱不是 nullable(如果可能的话)。 注意,一个更好的方法是使用 @NotNull 注释的Bean验证 规范。

让我们看看几个例子:

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

private String firstname; //persistent property

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

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

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

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

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

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

计数器,一个瞬变场, lengthInMeter ,一个方法注解为 @Transient ,将被忽略的冬眠。 名称, 长度, 和 FirstName 属性映射持久化和 急切地获取(默认为简单的属性)。 这个 detailedComment 属性值会懒洋洋地 从数据库中取出一旦懒惰财产的实体第一次访问。 通常你不需要懒惰的简单 属性(不要混淆延时关联获取)。 这个 建议的替代方法是使用投影功能以jp - ql的 (Java持久性查询语言)或标准查询。

JPA支持属性映射的基本类型支持 Hibernate(所有基本Java类型,各自的包装和 可序列化的类)。 Hibernate注释支持开箱即用 枚举类型映射到一个序数列(或储蓄的枚举 序数)或基于字符串列(拯救枚举字符串 表示):坚持表示,默认顺序, 可以覆盖通过吗 @Enumerated 注释所示 注意 财产 的例子。

在普通的Java api,显精密时间不是 定义。 当处理时态数据你可能想描述 预期的精度在数据库。 时态数据可以有 日期, 时间, 或 时间戳 精度(即实际日期,只有 时间,或两者)。 使用 @Temporal 注释 微调。

@Lob 表明该财产应 坚持一个Blob或Clob取决于属性类型: java sql clob , 字符[], char[] 和 java . lang. 字符串 将被持久化到一个Clob。 java sql blob , byte[], byte[] 和 可序列化的 式将被持久化到一个Blob。

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

@Lob
public byte[] getFullCode() {
```



```
    return fullCode;
}
```

如果属性类型实现 io可序列化的java 并不是一个基本类型, 如果属性不标注 @Lob , 然后Hibernate 可序列化的 类型是 使用。

5 1 4 1 1. 类型

你也可以手动指定一个类型使用 @org.hibernate.annotations.Type 和一些 如果需要参数。 @Type.type 可以是:

1. Hibernate的名字基本类型: 整数, 字符串、性格、日期、时间戳、浮子、二进制,可序列化的, 对象, blob 等。
2. 一个Java类的名称和一个默认的基本类型:
整数、浮点数、字符、java . lang。 字符串,java util日期, java . lang。 整数,java sql clob 等。
3. 这个名字的一个序列化的Java类。
4. 类名称的自定义类型: com.illflow.type.MyCustomType 等。

如果你不指定一个类型,Hibernate将使用反射 在指定的属性和猜正确的Hibernate类型。 Hibernate将尝试解释返回类的名称的 属性的getter使用,在秩序、规则2、3和4。

@org.hibernate.annotations.TypeDef 和 @org.hibernate.annotations.TypeDefs 允许你 申报类型定义。 这些注解可以被放置在 类或包级别。 注意,这些定义是全球的 会话工厂(即使当定义在类级别)。 如果 类型是用在一个单一的实体,可以把定义的 实体本身。 否则,建议把定义 在包级别。 在下面的示例中,当Hibernate 遇到一个属性的类 PhoneNumer ,它 代表持久性策略定制映射类型 PhoneNumberType 。 然而,属性属于 其他类,也可以代表他们的持久性策略 PhoneNumberType ,通过显式地使用 @Type 注释。

注意

包级别注释被放置在一个文件命名 包信息的java 在适当的包。 把你的注解在包的声明。

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

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

下面的示例展示了使用 参数 属性来定制 定义类型。

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

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

当使用复合用户类型,您将不得不表达 列定义。 这个 @Columns 一直 介绍了用于此目的。

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

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

```
} ...
```

5.1.4.1.2. 访问类型

默认情况下,访问类型的类层次结构的定义的位置 @id 或 @EmbeddedId 注释。如果这些注释在一个字段,那么只有字段是考虑持久性和 国家是通过现场访问。如果有注释是在 吸气,然后只有getter方法是考虑持久性和 状态是通过getter / setter。在实践中,工作得很好 和是推荐使用的。

注意

注释的放置在一个类层次结构有 是一致的(无论是字段或属性)能够 确定默认访问类型。建议坚持 一个注释放置在你的整个战略 应用程序。

然而在某些情况下,您需要:

- 力的接入类型实体层次结构
- 覆盖访问类型的一个特定的实体类 层次
- 覆盖访问类型的一个可嵌入的类型

最好的用例是使用一个可嵌入类由几个 实体可能不使用相同的访问类型。在这种情况下它是 更好的强迫访问类型的可嵌入类水平。

强制访问类型在一个给定的类,使用 @ access 注释显示如下:

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

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

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

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

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

    private hashCode; //not persistent
}
```

你也可以覆盖访问类型的一个属性 同时保持其他属性的标准。

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

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

在这个例子中,默认的访问类型 领域 除了 ordernumber 财产。注意,相应的 场,如果任何必须标记为 @Transient 或 瞬态。

@org.hibernate.annotations.AccessType

注释 @org.hibernate.annotations.AccessType 应该考虑弃用字段和属性访问。它 仍然是有用的但是如果你需要使用一个自定义的访问 类型。

5 1 4 1 3. 乐观锁

它有时是有用的,以避免增加版本号 即使一个给定的属性是脏(特别是集合)。 你 可以通过注释属性(或集合) @OptimisticLock(排除= true)。

更正式,指定更新这个属性不 需要采集的乐观锁。

5 1 4 1 4. 宣布列属性

列(s)用于属性映射可以定义使用 这个 @ column 注释。 用它来覆盖 默认值(参见JPA规范的更多信息 默认值)。 您可以使用 该注释的属性级别的 属性:











- ▶ 不带注释的所有
- ▶ 标注 @Basic
- ▶ 标注 version
- ▶ 标注 @Lob
- ▶ 标注 @Temporal

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

这个 名称 属性映射到 飞行的名字 列,这是不得为空,有一个 长度为50,不是可更新(使属性 不可变的)。

这个注释可用于常规属性 作为 @ id 或 version 属性。

```
@Column(
    name="columnName";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
    int length() default 255;
    int precision() default 0; // decimal precision
    int scale() default 0; // decimal scale
```

-  名称 (可选):列名 (默认为属性名)
-  独特的 (可选):设置一个独特的 在这列或不限制(默认错误)
-  Nullable (可选):设置专栏 为nullable(默认真实)。
-  可插入的 (可选):是否 列将insert语句的一部分(默认 真正的)
-  可更新的 (可选):是否 列的一部分更新语句(默认 真正的)
-  columnDefinition (可选):覆盖 sql DDL片段这个特定列(非 便携式)
-  表 (可选):定义目标 表(默认的主要表)
-  长度 (可选): 列的长度(默认255)
-  精密 (可选):列小数精度(缺省为0)
-  规模 (可选): 列十进制如果有用(缺省为0)

5 1 4 1 5. 公式

有时候,你想要的数据库做一些计算 你不是在JVM中,你也可以自己创造一些 虚拟列。 您可以使用一个SQL片段(aka公式)代替 映射属性到一个列。 这种属性是只读的 (它的价值是由你的公式计算片段)。

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

SQL片段可以复杂的你想要的,甚至 包括subselects。

5 1 4 1 6. 非注释的缺省属性












如果一个属性是不带注释的,下面的规则 应用:

- » 如果属性是一个类型,它被映射为 @Basic
- » 否则,如果属性的类型是注解为 @Embeddable,它被映射为@Embedded
- » 否则,如果属性的类型是 可序列化的,它被映射为 @Basic 在一列控股对象 在它的序列化版本
- » 否则,如果属性的类型是 java sql clob 或 java sql blob ,它被映射为 @Lob 与适当的 LobType

5 1 4 2. 属性映射与hbm xml

这个 <属性> 元素声明一个 持久的JavaBean样式属性的类。

```
<property
  name="propertyName"
  column="column_name"
  type="typename"
  update="true|false"
  insert="true|false"
  formula="arbitrary SQL expression"
  access="field|property|ClassName"
  lazy="true|false"
  unique="true|false"
  not-null="true|false"
  optimistic-lock="true|false"
  generated="never|insert|always"
  node="element-name|@attribute-name|element/@attribute|"
  index="index_name"
  unique_key="unique_key_id"
  length="L"
  precision="P"
  scale="S"
/>
```

-  名称 :属性的名称,一个 最初的小写字母。
-  列 (可选,默认了 属性名)的名称映射的数据库表列。 这也可以指定嵌套 <列> 元素(年代)。
-  类型 (可选):一个名字表明 Hibernate类型。
-  更新、插入 (可选,默认 真正的):指定映射的列 应包括在SQL 更新 和/或 插入 语句。 设置既 假 允许一个纯粹的“派生”属性的 值是初始化一些其他属性映射到 同一列(s),或由一个触发器或其他应用程序。
-  公式 (可选):一个SQL表达式 这定义了值 计算 财产。 计算属性没有列的映射 他们自己的。
-  访问 (可选,默认 财产):战略Hibernate使用 访问属性值。
-  懒惰 (可选,默认 假):指定该属性应该 被取出懒洋洋地当实例变量在第一次访问。 它需要构建时字节码插装。
-  独特的 (可选):使DDL 一代的一个独特的约束的列。 同时,允许 这是目标的一个 ref属性 。
-  过的非Null的 (可选):使DDL 一代的一个可为空特性约束的列。
-  乐观锁 (可选,默认 真正的):指定更新这个 财产做或不需要采集的乐观 锁。 换句话说,它决定了如果一个版本增量 应该发生在当这个属性是脏的。
-  生成 (可选,默认 从来没有):指定该属性值 实际生成的数据库。 看到讨论 [生成的属性](#) 更多 信息。

typename 可能是:

1. Hibernate的名字基本类型: 整数, 字符串、性格、日期、时间戳、浮子、二进制,可序列化的, 对象, blob 等。
2. 一个Java类的名称和一个默认的基本类型:
整数、浮点数、字符、 java . lang . 字符串, java util日期, java . lang . 整数, java sql clob 等。
3. 这个名字的一个序列化的Java类。
4. 类名称的自定义类型: com.illflow.type.MyCustomType 等。

如果你不指定一个类型, Hibernate将使用反思 指定的属性和猜正确的Hibernate类型。 Hibernate 将尝试解释返回类的名称的属性 吸气剂使用,在秩序、规则2、3和4。 在某些情况下,您将 需要 类型 属性。 例如,要 区分 hibernate日期 和 冬眠的时间戳 或者指定一个自定义 类型。

这个 访问 属性允许你控制 Hibernate如何在运行时访问属性。 默认情况下,冬眠 将调用属性的获取/设置一对。 如果你指定 访问=“字段” , Hibernate将绕过get / set 对和直接访问字段使用反射。 你可以指定 你自己的战略属性访问的命名的类 实现接口 org.hibernate.property.PropertyAccessor 。

一个强大的特性是派生属性。 这些属性是 根据定义只读。 属性值是计算在加载时间。 你申报的计算作为一个SQL表达式。 这就意味着 一个 选择 条款在SQL查询的子查询 装载一个实例:

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

你可以参考实体表不声明一个别名上 一个特定的列。 这将是 customerId 在 给定的例子。 您还可以使用嵌套的 <公式> 映射元素如果你不想 使用属性。

是5.1.5。 嵌入对象(aka组件)

可嵌入的对象(或组件)是对象的属性 被映射到相同的表作为拥有实体的表。 组件 可以,反过来,声明自己的属性,组件或 集合

可以声明一个嵌入式组件在一个实体 甚至覆盖它的列映射。 组件类必须 带注释的类级别的 @Embeddable 注释。 它可以覆盖列映射的嵌入式 对象为一个特定的实体使用 @Embedded 和 @AttributeOverride 注释在相关 属性:

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
    Address homeAddress;

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

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

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

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

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

一个可嵌入对象继承了其拥有实体的访问类型 (请注意,您可以覆盖,使用 @ access 注释)。

这个人 实体有两个组件属性, homeAddress 和 队员。 homeAddress 房地产没有注释,但是 Hibernate会猜,它是一个持久性组件通过寻找 这个 @Embeddable 注释在地址类。 我们 还覆盖映射的列名称(来 bornCountryName) @Embedded 和 @AttributeOverride 为每个映射属性的注释 国家。 正如您可以看到的, 国家 也是一个嵌套的组件的地址, 再次使用Hibernate和JPA违约的自动识别。 覆盖 列的嵌入对象的嵌入对象是通过虚线 表达式。

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

Hibernate注释支持一些不那么明确 支持JPA规范。 你可以标注嵌入对象 与 @MappedSuperclass 注释使 超类属性持久 (见 @MappedSuperclass 以获得更多信息)。

您还可以使用一个嵌入式协会注释对象 (即 @OneToOne , @ManyToOne , onetomany 或 @ManyToMany)。 到 覆盖关联字段可以使用 @AssociationOverride 。

如果你想拥有相同的可嵌入的对象类型的两倍 相同的实体,列名违约将无法正常工作几个 嵌入对象将共享相同的组列。 在普通的JPA,你 需要覆盖至少一组列。 Hibernate,然而,允许 你来增强默认命名机制通过 namingStrategy 接口。 你可以写一个策略,防止名称冲突在这种情况下。 DefaultComponentSafeNamingStrategy 就是一个例子 这个。









如果一个属性的嵌入式对象分回拥有 实体,注释的 @Parent 注释。 Hibernate将确保这个属性是装载的 实体引用。

在XML,使用 <组件> 元素。

```
<component
  name="propertyName"
  class="className"
  insert="true|false"
  update="true|false"
  access="field|property|ClassName"
  lazy="true|false"
  optimistic-lock="true|false"
  unique="true|false"
  node="element-name|. "
>

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

.....
</component>

-  名称 :属性的名称。
-  类 (可选,默认了 属性类型取决于反射):组件的名称 (孩子)类。
-  插入 :做映射的列中出现 SQL 插入 吗?
-  更新 :做映射的列中出现 SQL 更新 吗?
-  访问 (可选,默认 财产):战略Hibernate使用 访问属性值。
-  懒惰 (可选,默认 假):指定该组件应该 当实例变量获取懒洋洋地在第一次访问。 它 需要构建时字节码插装。
-  乐观锁 (可选,默认 真正的):指定更新该组件 要么做或不需要采集的乐观锁。 它 确定如果一个版本增量应该发生在当 这个属性 是脏的。
-  独特的 (可选,默认 假):指定一个唯一约束 存在在所有列的映射组件。

孩子 <属性> 标签地图属性 孩子的类表列。

这个 <组件> 元素允许 <父> 子元素,地图的一个属性 组件类作为参考回包含实体。

这个 <动态组件> 元素允许 地图 将被映射为一个组件的属性 名称是指键的映射。 看到 9.5节,“动态组件” 为更多的信息。 这个特性是 不支持的注解。

5—6. 继承策略

Java语言支持多态性:一个类可以继承 从另一个。 几个策略可能存在一个类 层次结构:

- ▶ 单表每个类层次的策略:一个表 主机的所有实例的类层次结构
- ▶ 加入了子类的策略:一个表每个类和子类 现在和每个表存在特定于给定的属性 子类。 实体的状态然后存储在它的 相应的 类表和所有它的超类
- ▶ 按类表策略:每个具体类一个表和 子类是现在和每个表持续的属性 类和它的超类。 实体的状态被存储 完全在专用表为其 类。

5 1 6 1. 单表每个类层次的策略

使用这种方法的所有子类的属性在一个 鉴于映射的类层次结构存储在一个单一的表。

每个子类宣称自己的持久性属性和 子类。 版本和id属性被认为是继承 根类。 每个子类在层次结构中必须定义一个独特的 鉴别器值。 如果不指定这个值,则完全限定 使用Java类名称。





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

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

在hbm. xml,因为表每个类层次映射策略,这个 <子类> 使用声明。 对于 示例:

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

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

-  名称 :完全限定类名 子类。
-  discriminator值 (可选的— 默认为类名):一个值区分开来 个人子类。
-  代理 (可选):指定一个类或 接口用于懒初始化代理。
-  懒惰 (可选,默认 真正的):设置 懒= " false " 禁用使用懒惰 抓取。

对于信息的继承映射看到 [第十章, 继承映射](#)。

5 1 6 1 1. 鉴别器

鉴别器所需的多态持久性使用 表每个类层次的映射策略。 它声明一个 鉴别器列的表。 discriminator包含 标记值,告诉持久层什么子类 实例化特定行。 Hibernate核心支持 阶段限制组类型鉴别器列: 字符串, 字符, 整数, 字节, 短, 布尔, 是的没有, 真的假。

使用 @DiscriminatorColumn 定义 discriminator以及鉴别器类型。

注意

DiscriminatorType的枚举 用于 javax.persistence.DiscriminatorColumn 只有 包含值 字符串, char 和 整数 哪一个 意味着并不是所有Hibernate支持类型可通过 这个 @DiscriminatorColumn 注释。

您还可以使用 @DiscriminatorFormula 在SQL来表达一个 虚拟鉴别器列。 这是特别有用的时候 鉴别器值可以提取一个或多个列的 表。 两 @DiscriminatorColumn 和 @DiscriminatorFormula 要设置吗 根实体(一次/持久化的层次结构)。

@org.hibernate.annotations.DiscriminatorOptions 可以选择指定Hibernate具体鉴别器 选择,都不是标准化在JPA。 可用的选项是 力 和 插入。 这个 力 属性是有用的,如果表包含 行以 “额外” 鉴别器值不映射到一个 持久化类。 这可能例如发生在处理一个 遗留数据库。 如果 力 设置为 真正的 Hibernate将指定允许的 discriminator值 选择 查询,甚至 当检索所有实例的根类。 第二个选项—— 插入 ——告诉Hibernate是否 包含discriminator在SQL 插入。 通常列应该的部分 插入 声明,但如果你鉴别器列也属于一个映射 复合标识符必须设置这个选项 假。

提示

还有一个 @org.hibernate.annotations.ForceDiscriminator 注释是弃用自版本3.6。 使用 @DiscriminatorOptions 相反。






最后,使用 @DiscriminatorValue 在 每一个类层次结构的指定值存储在 鉴别器列对于一个给定的实体。 如果你不设置 @DiscriminatorValue 在一个类,完全 使用完全限定类名。

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

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

在hbm. xml, <鉴别器> 元素用于定义discriminator或 公式:

```
<discriminator
    column="discriminator_column"
    type="discriminator_type"
    force="true|false"
    insert="true|false"
    formula="arbitrary sql expression"
/>
```

-  列 (可选,默认 类):鉴别器的名称 列。
-  类型 (可选,默认 字符串):一个名字,表示 Hibernate类型
-  力 (可选,默认 假): “部队” Hibernate来指定 允许鉴别器值,甚至当检索所有 根类的实例。
-  插入 (可选,默认 真正的):这个设置 假 如果你的鉴别器列也属于一个映射 复合标识符。 它告诉Hibernate不包括 列在SQL 插入。
-  公式 (可选):任意SQL 表达时所执行一个类型必须被评估。 它允许基于内容的歧视。

实际值的甄器列指定的 discriminator值 属性的 <类> 和 <子类> 元素。

这个 公式 属性允许您声明 一个任意SQL表达式,用于评价类型 连续的。 例如:

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

5 1 6 2. 加入子类策略

每个子类也可以映射到自己的表。这是称为表每个子类映射策略。一种遗传状态是通过加入检索表的超类。一个鉴别器列不需要这种映射策略。每个子类必须，然而，声明一个表列持有对象标识符。这个这个表的主键也是一个外键到超类。表和描述 @PrimaryKeyJoinColumn 年代或 <键> 元素。

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

    ...
}

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

注意





表名仍然默认为非限定类名。如果 @PrimaryKeyJoinColumn 没有设置,那么 主键/外键列被假定为具有相同的名称 作为主键列的基本表的 超类。

在hbm. xml,使用 <加入子类> 元素。例如:

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

  <key .... >

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

-  名称 :完全限定类名 子类。
-  表 :子类的名称 表。
-  代理 (可选):指定一个类或 接口用于懒初始化代理。
-  懒惰 (可选,默认 真正的):设置 懒= " false " 禁用使用懒惰 抓取。

使用 <键> 元素来声明 主键/外键列。映射在开始的 章将被重写为:

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

<hibernate-mapping package="eg">

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

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

</hibernate-mapping>
```


对于信息的继承映射看到 [第十章, 继承映射](#)。

5 1 6 3. 按类表策略





第三个选项是地图只有具体类的 继承层次结构表。这被称为 每个具体类一张表的策略。每个表定义所有持久 州的类,包括继承的状态。在冬眠,它 没有必要显式映射这样的继承层次结构。你 可以映射每个类作为一个单独的实体根。然而,如果你希望使用 多态关联(如一个协会的超类 你的层次),您需要使用联盟子类映射。

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

或在hbm.xml:

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

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

-  名称 :完全限定类名 子类。
-  表 :子类的名称 表。
-  代理 (可选):指定一个类或 接口用于懒初始化代理。
-  懒惰 (可选,默认 真正的):设置 懒= " false " 禁用使用懒惰 抓取。

没有鉴别器列或键列是需要这个 映射策略。

对于信息的继承映射看到 [第十章, 继承映射](#)。

5 1 6 4. 从超类继承属性

这有时是有用的共享共同的属性通过一个 技术或商业超类没有包括它作为常规 映射实体(即没有特定的表,这个实体)。为此目的 你可以将它们映射作为 @MappedSuperclass 。

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

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

在数据库中,这种层次结构将被表示为一个 秩序 表有 id , lastupdate 和 lastUpdater 列。 嵌入的超类属性映射被复制到 他们的实体子类。记住,可嵌入的超类是 没有层次结构的根虽然。

注意

从超类没有映射属性作为 @MappedSuperclass 被忽略。

注意

默认的访问类型(字段或方法)使用,除非你 使用 @ access 注释。

注意

相同的概念可以应用到 @Embeddable 对象持久化属性 他们的超类。 你还需要使用 @MappedSuperclass 这样做(这是不应当的 认为是一个标准的EJB3特性虽然)

注意

它是允许标记一个类 @MappedSuperclass 在中间的映射 继承层次结构。

注意

任何类层次结构中的非注释 @MappedSuperclass 也 entity 将被忽略。

你可以覆盖父类中定义的列的实体 根实体级使用 @AttributeOverride 注释。

```
@MappedSuperclass
public class FlyingObject implements Serializable {

    public int getAltitude() {
        return altitude;
    }

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

    @ManyToOne
    public PropulsionType getPropulsion() {
        return metricAltitude;
    }
    ...
}

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

这个 高度 属性将坚持一个 盛名高度 列的表 平面 和推进协会将 物化在 盛名推进第 外键 列。

您可以定义 @AttributeOverride (s)和 @AssociationOverride (s) entity 类, @MappedSuperclass 类和属性指向 一个 @Embeddable 对象。

在hbm. xml,简单的地图的属性的超类 <类> 元素实体的需要 继承它们。

5 1 6 5. 一个实体映射到多个表

而不建议一个新的模式,一些遗留数据库 强迫你来映射一个单一的实体在几个表。

使用 @SecondaryTable 或 @SecondaryTables 类级别注释。 到 表达一个列在一个特定的表,可以使用 表 参数的 @column 或 @JoinColumn 。

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

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

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

    public String getName() {
        return name;
    }

    @Column(table="Cat1")
```

```

public String getStoryPart1() {
    return storyPart1;
}

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

```

在这个例子中, 名称 将在 MainCat 。 storyPart1 将在 Cat1 和 storyPart2 将在 Cat2 。 Cat1 将加入到 MainCat 使用猫id 作为一个 外键, Cat2 使用 id (即相同的列名称, MainCat id列 有)。 加上独特的约束 storyPart2 已经 被设置。

还有额外的调整可以通过 @org.hibernate.annotations.Table 注释:

- ▶ 取 :如果设置加入,默认的, Hibernate将使用内连接检索二次表 定义为一个类或其超类和一个外部联接为一个 二次表定义一个子类。 如果设置为 选择 然后Hibernate将使用一个顺序 选择一个二次表上定义一个子类,它会 只有一行发行原来代表的一个实例 子类。 内连接仍将被用来检索二次 定义的类和它的超类。
- ▶ 逆 :如果这是真的,Hibernate将不会尝试 插入或更新属性定义为该连接。 默认 为false。
- ▶ 可选 :如果启用(默认的), Hibernate将插入一行只有在属性定义为本 加入非空,永远是使用一个外部连接来检索 属性。
- ▶ ForeignKey :定义外键名称 一个二次表指向主桌。

确保使用二次表名称 appliesto 财产

```

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

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

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

    public String getName() {
        return name;
    }

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

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

```

在hbm. xml,使用 <加入> 元素。

```

<join
    table="tablename"
    schema="owner"
    catalog="catalog"
    fetch="join|select"
    inverse="true|false"
    optional="true|false">

    <key ... />


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


```


 表 :加入的名称 表。

 模式 (可选):覆盖模式 名指定的根 < hibernate映射> 元素。

 目录 (可选):覆盖 目录名称指定的根 < hibernate映射> 元素。

 取 (可选,默认 加入):如果设置为 加入, 默认情况下,Hibernate将使用内连接检索 <加入> 定义为一个类或其 超类。 它将使用一个外部连接为一个 <加入> 定义一个子类。 如果设置为 选择 然后Hibernate将使用一个顺序 选择一 <加入> 定义在 子类。 这将发布只有一行表示一个 子类的实例。 内连接仍将被用于 检索 <加入> 类定义的 和它的超类。

 逆 (可选,默认 假):如果启用,Hibernate不会插入 或更新属性定义为该连接。

 可选 (可选,默认 假):如果启用,Hibernate将插入一个 行只有在属性定义为这个连接是null。 它 总是使用一个外部连接来检索属性。

例如,对一个人的地址信息可以被映射到一个 单独的表,同时保留值类型语义对于所有 属性:

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

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

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

这个特性常常只有有用的遗留数据模型。我们推荐表少比类和一个细粒度的域模型。然而,它是有用的对于切换继承映射策略在一个层次,因为后来解释说。

5-7. 映射一个对一个,另一个许多联想

链接到另外一个实体,你需要地图的协会 房地产作为一个协会。在关系模型中,您可以 或者使用一个外键或一个关联表,或(有点不太常见的) 共享相同的主键值的两个实体之间。

标记一个协会,或者使用 @ManyToOne 或 @OneToOne 。

@ManyToOne 和 @OneToOne 有一个参数命名 targetEntity 描述 目标实体的名字。你通常不需要这个参数自 默认值(属性的类型,存储协会) 好的在几乎所有情况下。然而这是有用的,当你想使用 接口作为返回类型而不是一般的实体。

设置的值 级联 属性到任何 有意义的值比没有将传播某些操作 关联的对象。有意义的值分为三种 类别。

1. 基本操作,包括: 坚持,合并, 删除,保存更新,驱逐,复制、锁和 刷新 ;
2. 特殊值: 删除孤儿 或 所有 ;
3. 逗号分隔的组合操作名称: 级联 = "持续、合并、驱逐" 或 级联 = ",删除孤儿" 。看到 11.11节, "过渡持久性" 对于一个完整的解释。注意 这单值的多对一关联不支持孤儿 删除。

默认情况下,单点关联热切引进JPA 2。你可以将其标记为懒洋洋地获取使用 @ManyToOne(fetch = FetchType.LAZY) 在这种情况下 Hibernate将代理协会和负载状态的时候 达到相关的实体。你可以强迫不使用Hibernate的代理 通过使用 @LazyToOne(没有代理) 。在这种情况下, 房地产是拿来懒洋洋地当实例变量在第一次访问。这需要构建时字节码插装。懒 = " false " 指定该协会总是急切地获取。

用默认JPA选项,单端关联被加载 与后续的选择如果设置为 懒惰, 或一个SQL 加入用于 急切 协会。不过你可以 调整抓取策略, 即如何通过抓取数据 @Fetch 。 FetchMode 可以 选择 (一个选择时触发协会 需要加载)或 加入 (使用一个SQL连接到负载 协会同时加载业主单位)。加入 覆盖任何懒属性(一个协会的装入 加入 战略不能懒)。

5 1 7 1. 使用一个外键或一个关联表

一个普通的关联到另一个持久化类声明 使用

- » @ManyToOne 如果几个实体可以 指向目标的实体
- » @OneToOne 如果只有一个单一的实体可以 指向目标的实体

和一个外键引用一个表的主键 列(s)的目标表。

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

这个 @JoinColumn 属性是可选的, 默认值(s)是串联关系的名称 在主人身边, _ (下划线),这个名字的 主键列在拥有侧。 在这个例子 公司id 因为属性的名字是 公司 和列id的公司 id 。

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

public interface Company {
    ...
}
```

你也可以映射到一个协会通过一个协会 表。 本协会表所描述的 @JoinTable 注释将包含一个外键 引用返回实体表(通过 @JoinTable.joinColumns)和一个外键 引用目标实体表(通过 @JoinTable.inverseJoinColumns)。

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

注意

您可以使用一个SQL片段来模拟一个物理连接列 使用 @JoinColumnOrFormula / @JoinColumnOrFormulas 注释(就像 您可以使用一个SQL片段来模拟一个属性列通过 @ formula 注释)。

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

你可以标记一个协会作为强制使用 可选= false 属性。 我们建议使用Bean 验证的 @NotNull 注释作为一个更好的 替代然而。 因此,外键列(s) 将被标记为nullable(如果可能的话)。

当Hibernate不能解决协会因为 预计相关的元素不是在数据库(错误id的 协会列),会抛出一个异常。 这可能是 不方便维护模式遗留和严重。 你可以问 Hibernate忽略这样的元素,而不是引发一个异常 使用 @NotFound 注释。

例5.1. @NotFound注释

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

有时你想委托给你的数据库的删除 当一个给定的实体级联删除。 在这种情况下Hibernate 生成一个级联删除约束在数据库级别上。

例5.2. @OnDelete注释

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

外键约束,而产生的冬眠,有一个 相当难以理解的名称。 你可以覆盖约束名称使用 @ForeignKey 。

例5.3. @ForeignKey注释

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

```

public Parent getParent() { ... }
...
}

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

```

有时候,你想链接到另外一个实体不是 目标实体的主键,但由不同独特的键。 你可以 实现,通过引用独特的键列(s) @JoinColumn.referenceColumnName 。

```

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

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

```
















这不是鼓励然而,应该保留遗留 映射。

在hbm. xml映射一个协会是相似的。 主要 区别在于 @OneToOne 被映射为 <多对一的独特= " true " / > ,让我们深入这个主题。

```

<many-to-one
    name="propertyName"
    column="column_name"
    class="ClassName"
    cascade="cascade_style"
    fetch="join|select"
    update="true|false"
    insert="true|false"
    property-ref="propertyNameFromAssociatedClass"
    access="field|property|ClassName"
    unique="true|false"
    not-null="true|false"
    optimistic-lock="true|false"
    lazy="proxy|no-proxy|false"
    not-found="ignore|exception"
    entity-name="EntityName"
    formula="arbitrary SQL expression"
    node="element-name|@attribute-name|element/@attribute|"
    embed-xml="true|false"
    index="index_name"
    unique_key="unique_key_id"
    foreign-key="foreign_key_name"
/>

```

-  名称 :属性的名称。
-  列 (可选)的名字 外键列。 这也可以指定嵌套 <列> 元素(年代)。
-  类 (可选,默认了 属性类型取决于反射)的名字 相关的类。
-  级联 (可选):指定 操作应该从父对象级联到 关联的对象。
-  取 (可选,默认 选择):选择外连接抓取之间 或顺序选择抓取。
-  更新、插入 (可选,默认 真正的):指定映射的列 应包括在SQL 更新 和/或 插入 语句。 设置既 假 允许一个纯粹的“派生” 协会 初始化的值是来自另一个属性映射到吗 相同的列(s),或由一个触发器或其他应用程序。
-  ref属性 (可选):名称 属性关联的类,它是与以外国 关键。 如果未指定,主键关联的类 是使用。
-  访问 (可选,默认 财产):战略Hibernate使用 访问属性值。
-  独特的 (可选):使DDL 一代的一个独特的约束为外键列。 通过 允许这是目标的一个 ref属性 ,你可以使协会 多重性一对一。
-  过的非null的 (可选):使DDL 一代的一个可为空特性约束的外键 列。
-  乐观锁 (可选,默认 真正的):指定更新这个 财产做或不需要采集的乐观 锁。 换句话说,它决定了如果一个版本增量 应该发生在当这个属性是脏的。
-  懒惰 (可选,默认 代理):默认情况下,单点联系 是代理。 懒=“没有代理” 指定 房地产应该拿来懒洋洋地当实例变量 在第一次访问。 这需要构建时字节码 仪表。 懒= " false " 指定 该协会将总是急切地获取。
-  没有找到 (可选,默认 异常):指定如何外键 引用缺失行处理。 忽略 会对一个失踪的行作为一个空吗 协会。
-  实体名称 (可选):实体名称 关联的类。
-  公式 (可选):一个SQL表达式 这定义了值 计算 外键。

设置的值 级联 属性 任何有意义的值 没有 将传播 某些操作相关的对象。 有意义的值 分为三个类别。 首先,基本操作,包括:

坚持、合并、删除、保存更新、驱逐、复制、锁 和刷新 ;第二,特殊值: 删除孤儿 ;第三, 所有 逗号分隔的组合操作名称: 级联= "持续、合并、驱逐" 或 级联= "删除孤儿" 。看到 11.11节, "过渡持久性" 对于一个完整的解释。 注意, 单值、多对一、一对一,联想不支持 孤儿删除。

这是一个典型的 多对一 声明:

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

这个 ref属性 属性只能 用于映射遗留数据,一个外键是指一种独特的 键关联的表除了主键。 这是一个 复杂和令人困惑的关系模型。 例如,如果 产品 类有一个独一无二的序列号,是 没有主键。 这个 独特的 属性控制 Hibernate的DDL生成与 SchemaExport工具。

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

然后映射为 OrderItem 可能 使用:

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

这不是鼓励,然而。

如果所引用的唯一键由多个属性的 相关的实体,你应该地图所引用的属性里 一个名叫 <属性> 元素。

如果所引用的唯一键的性质是一个组件,你 可以指定一个属性路径:

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

5 1 7 2. 共享主键和关联的实体

第二种方法是确保一个实体及其相关的 实体共享相同的主键。 在这种情况下,主键列 也是一个外键,没有额外的列。 这些协会 总是一个。

例5.4. 一个对一个协会

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

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


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










注意

很多人弄糊涂了,这些主键建立一个对一个 协会。 他们只能延迟加载的如果Hibernate知道 另一边的协会是永远存在的。 显示到 Hibernate,是这样的情况,使用 @OneToOne(可选= false) 。

在hbm. xml,使用下面的映射。

```
<one-to-one
name="propertyName"
class="ClassName"
cascade="cascade_style"
constrained="true|false"
fetch="join|select"
property-ref="propertyNameFromAssociatedClass"
access="field|property|ClassName"
formula="any SQL expression"
lazy="proxy|no-proxy|false"
entity-name="EntityName"
node="element-name|@attribute-name|@attribute|"
embed-xml="true|false"
foreign-key="foreign_key_name"
/>
```

 名称 :属性的名称。

-  类 (可选,默认了 属性类型取决于反射的名字 相关的类。
-  级联 (可选):指定 操作应该从父对象级联到 关联的对象。
-  约束 (可选):指定 一个外键约束的映射表的主键 和引用表关联的类。 这个选项 影响顺序 save() 和 删除() 级联,并确定吗 该协会可以代理。 它也使用这个模式 导出工具。
-  取 (可选,默认 选择):选择外连接抓取之间 或顺序选择抓取。
-  ref属性 (可选):名称 属性关联的类,是加入到主 这个类的键。 如果未指定,则主键的 使用关联的类。
-  访问 (可选,默认 财产):战略Hibernate使用 访问属性值。
-  公式 (可选):几乎所有的 一对一关联映射到主键的拥有 实体。 如果不是这种情况,那么您可以指定另一个列, 列或表达式连接使用一个SQL公式。 看到 org.hibernate.test.onetooneformula 对于一个 的例子。
-  懒惰 (可选,默认 代理):默认情况下,单点联系 是代理。 懒= "没有代理" 指定 房地产应该拿来懒洋洋地当实例变量 在第一次访问。 它需要构建时字节码 仪表。 懒= " false " 指定 该协会将总是急切地获取。 注意 ,如果 约束= " false ",代理是 不可能和Hibernate会急切地获取 协会 。
-  实体名称 (可选):实体名称 关联的类。

主键关联不需要额外的表列。 如果 两行相关的协会,那么这两个表行分享 相同的主键值。 联系两个对象的主键 协会,确保他们 被分配相同的标识符 值。

主键关联,添加下面的映射 员工 和 人 分别为:

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

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

确保主键的相关行在 人 和 员工的表都是平等的。 你使用一个特殊的Hibernate标识符 生成策略称为 外国 :

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

一个新保存的实例 人 分配 相同的主键值 员工 实例 提到与 员工 性质 人 。

5 1 8. 自然id

虽然我们建议使用代理键作为主键, 你应该试着识别自然键所有实体。 一个自然的关键 是一个属性或属性的组合,是独特的和 非空。 它也是不变的。 地图的属性,自然键作为 @NaturalId 或者他们内部的地图 <自然id > 元素。 Hibernate将生成 必要的 惟一键,可为空特性约束,因此, 你的映射将更具可读性。

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

    @NaturalId
    @ManyToOne
    private State state;

    @NaturalId
    private String ssn;
    ...
}

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

或在XML,

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

建议您实现 equals() 和 hashCode() 比较自然的关键属性 的实体。

这种映射并不打算使用的实体,有 自然主键。

» 可变 (可选,默认 假):默认情况下,自然标识符属性 被认为是不可改变的(常数)。

5 1 9. 任何

还有一个类型的属性映射。这个 @Any 映射定义了一个多态的协会 类从多个表。这种类型的映射需要超过 一列。第一列包含类型关联的实体。剩下的列包含标识符。这是不可能的 指定一个外键约束对于这种协会。这是 不是通常的方法映射多态关联,你应该使用 这只有在特殊情况下。例如,对于审计日志、用户会话 数据等。

这个 @Any 注释描述列 抱着元数据信息。连接元数据的值 信息和实际的实体类型, @AnyDef 和 @AnyDefs 使用注释。这个 变型 属性允许 应用程序指定一个自定义类型,地图数据库列 值来持久化类,有标识符类型的属性 指定 idType 。你必须指定映射 从价值值的 变型 类的名字。

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

注意, @AnyDef 可以mutualized和 重用。建议将它作为一个包元数据在这 案例。

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

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





恒的。xml等价是:



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

注意

你不能mutualize元数据在hbm。xml可以在 注释。

```
<any
  name="propertyName"
  id-type="idtypename"
  meta-type="metatypename"
  cascade="cascade_style"
  access="field|property|ClassName"
  optimistic-lock="true|false"
>
  <meta-value ... />
  <meta-value ... />
  .....
  <column .... />
  <column .... />
  .....
</any>
```






-  名称 :属性名称。
-  id类型的 :标识符类型。
-  元类型 (可选,默认 字符串):任何类型,是允许的 鉴频器的映射。
-  级联 (可选,默认 没有):级联风格。

-  访问 (可选,默认 财产):战略Hibernate使用 访问属性值。
-  乐观锁 (可选,默认 真正的)指定更新这个属性 要么做或不需要采集的乐观锁。 它 定义一个版本增量应该发生是否如果该属性 是脏的。

5—10. 属性

这个 <属性> 元素允许 定义一个命名,逻辑分组的类的属性。 最重要的使用是可以构造组合 的属性的目标 ref属性 。 它 也是一个方便的方式来定义一个多列唯一约束。 对于 示例:

```
<properties
  name="logicalName"
  insert="true|false"
  update="true|false"
  optimistic-lock="true|false"
  unique="true|false"
>
  <property ...../>
  <many-to-one .... />
  .....
</properties>
```

-  名称 :的逻辑名称分组。 这是 不 一个实际的属性名。
-  插入 :做映射的列中出现 SQL 插入 吗?
-  更新 :做映射的列中出现 SQL 更新 吗?
-  乐观锁 (可选,默认 真正的)指定更新这些 属性要么做或不需要收购 乐观锁。 它决定了如果一个版本增量应该发生当这些属性都脏了。
-  独特的 (可选,默认 假):指定一个唯一约束 存在在所有的映射组件。

例如,如果我们有以下 <属性> 映射:

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

你可能有一些遗留数据协会,是指这个 唯一键的 人 表,而不是到 主键:

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

注意

当使用注释的映射策略,这样的构造 没有必要为一个列之间的绑定和其相关的列 在相关的表格是直接完成

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

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

使用这外面的上下文映射遗留数据不是 推荐。

5 1 11. 一些hbm。xml特异性

恒的。xml结构有一些特异性自然不存在 当使用注释,让我们简要描述它们。

5 1 11 1. Doctype

所有XML映射应该宣布doctype显示。实际的 DTD可以发现上面的URL,在目录 hibernate x x x / src / org/hibernate ,或在 hibernate3.jar 。Hibernate将永远寻找 在它的类路径第一。DTD 如果你经验查找的DTD使用 一个网络连接,检查DTD声明反对的内容 你的类路径。

5 1 11 1 1. EntityResolver

Hibernate将首先尝试解决它的类路径中的dtd。它是通过注册一个自定义的 org.xml.sax.entityresolver 实现与 它使用的 SAXReader读取xml文件。这个习俗 EntityResolver systemId承认两个不同 名称空间:

- » 一个 hibernate命名空间 是公认的 每次解析器遇到systemId开始 http://www.hibernate.org/dtd/ 。这个解析器 试图解决这些实体通过类加载器,加载类的冬眠。
- » 一个 用户名称空间 被公认每当 这个解析器遇到systemId使用 类路径:// URL协议。解析器将 试图解决这些实体通过 (1)当前线程 上下文类加载器和(2)的类加载器加载了 Hibernate类。

以下是一个例子,利用用户 命名空间:

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








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

在 类型xml 是一个资源 你的域名 包和包含一个定义 typedef 。

5 1 11 2. hibernate映射

这个元素有多个可选的属性。这个 模式 和 目录 属性 指定表指在这个映射属于命名 模式和/或目录。如果他们被指定,将表名 合格的由给定的模式和目录名称。如果他们失踪,将不合格的表名。这个 默认级联 属性指定级联风格应该假定为 属性和集合,不指定 级联 属性。默认情况下,汽车进口 属性允许您使用不合格 类名的查询语言。

```
<hibernate-mapping
  schema="schemaName"
  catalog="catalogName"
  default-cascade="cascade_style"
  default-access="field|property|ClassName"
  default-lazy="true|false"
  auto-import="true|false"
  package="package.name"
/>
```

-  模式 (可选):名称 数据库模式。
-  目录 (可选):名称 数据库目录。
-  默认级联 (可选,默认 没有):一个默认的级联风格。
-  默认访问 (可选,默认 财产):战略Hibernate应该使用 访问所有的属性。它可以是一个自定义的实现的 PropertyAccessor 。
-  默认懒惰 (可选,默认 真正的):默认值为未指定的 懒惰 类的属性和收集 映射。
-  汽车进口 (可选,默认 真正的):指定是否我们可以使用 不合格的名称的类在这个映射查询 语言。
-  包 (可选):指定一个包 前缀用于限定的类名的映射 文档。







如果你有两个相同的持久化类不合格 的名字,你应该设定 汽车进口= " false "。 一个 异常结果如果试图将两类分配到相同的 "进口" 的名字。

这个 hibernate映射 元素允许您 巢几个持续 <类> 的映射,因为 如上所示。 然而,它是良好的实践(和预期的一些 工具)来映射 只有一个持久化类,或者一个类 层次结构,在一个映射文件和名称后持久 超类。 例如, 猫hbm.xml, 狗hbm.xml, 或者如果使用 继承, 动物hbm.xml。

5 1 11 3. 关键

这个 <键> 元素是精选几 次在本指南。 看来任何父映射 元素定义了一个连接到一个新表引用主键 原始的表。 它还定义了外 键在加入 表:

```
<key
  column="columnname"
  on-delete="noaction|cascade"
  property-ref="propertyName"
  not-null="true|false"
  update="true|false"
  unique="true|false"
/>
```

-  列 (可选)的名字 外键列。 这也可以指定嵌套 <列> 元素(年代)。
-  在删除 (可选,默认 再):指定是否外键 约束数据库级的级联删除启用。
-  ref属性 (可选):指定 外键是指列不是主键 原始的表。 它提供遗留数据。
-  过的非Null的 (可选):指定 外键列不为空。 这是隐含每当 外键也主键的一部分。
-  更新 (可选):指定 外键不应该被更新。 这是隐含每当 外键也主键的一部分。
-  独特的 (可选):指定 外键应该有一个独特的约束。 这是隐含 每当外键也主键。

对系统在删除性能很重要,我们建议 所有的键应该被定义 在删除="级联"。 Hibernate使用 数据库级 在级联删除 约束,而 不是许多个人 删除 语句。 是 注意,该特性会绕过Hibernate的常用乐观锁 版本数据的策略。



这个 过的非Null的 和 更新 属性是有用,当一个单向一对多的映射 协会。 如果你地图一个单向一对多关联到一个 非空的外键, 你必须 宣布 键列使用 <关键 过的非null = " true " > 。

5 1 11 4. 进口

如果你的应用程序有两个相同的持久化类 的名字,和你不想指定完全限定的包名 在Hibernate查询,类可以 "进口",而是明确 比依赖 汽车进口 = " true "。 你也可以 进口类和接口不明确的映射:

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

```
<import
  class="ClassName"
  rename="ShortName"
/>
```

-  类 :完全限定类名 任何Java类。
-  重命名 (可选,默认了 不合格类名):一个名称,可以查询中使用 语言。

注意

这个功能是独一无二的hbm。 xml和不支持 注释。

5 1 11 5. 列和公式元素

映射元素接受 列 属性将或者接受 <列> 子元素。 同样的, <公式> 是一种替代吗 公式 属性。 例如:

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

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

大部分的属性列 提供一个 裁剪的方法在自动模式生成的DDL。这个 读 和 写 属性允许 您指定自定义SQL,Hibernate将使用访问 列的值。更多详情请查看讨论 [列读和写 表达式](#)。

这个 列 和 公式 元素甚至可以组合在同一个属性或协会 映射来表达,例如,异国情调的加入条件。

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

5.2. Hibernate类型

5.2.1. 实体和值

相对于持久性服务,Java语言 对象分为两组:

一个 实体 存在独立于任何其他 对象持有引用实体。与之形成对比的是,通常的 Java模型,未引用的对象被垃圾收集了。实体 必须显式保存和删除。保存和删除,但是,可以 是 级联 从父母的孩子。实体 这是不同的模型对象持久化的ODMG通过 可达性和更密切的对应如何应用对象 通常用于大型系统。实体支持循环和共享 引用。他们也可以进行版本控制。

一个实体的持久状态包括引用其他 实体和实例的 价值 类型。值 原语;集合(没有什么是在一个集合),组件 和某些不可变对象。不像实体、价值观尤其是 集合和组件,是 坚持和 通过可删除。因为值对象和原语 保存和删除以及其包含的实体,他们不能 独立版本。值没有独立的身份,所以他们 不能被共享的两个实体或集合。

直到现在,我们一直在使用术语“持久化类”表示 实体。我们将继续这样做。并不是所有的用户定义的类 持续的状态,然而,是实体。一个 组件 是一个用户定义的类与价值 语义。Java属性的类型 以 也有值语义。鉴于这种定义,所有类型(类) 提供的JDK有价值类型语义在Java中,虽然 用户定义类型可以映射与实体或值类型语义。这个决定取决于应用程序开发人员。一个 实体类 域模型通常会有共享引用一个实例 这个类的,而组成或聚合通常转化成一个 值类型。

我们将重新在此引用两个概念 指南。

面临的挑战是映射Java类型系统,开发人员的 定义的实体和值类型的SQL /数据库类型系统。两个系统之间的桥梁是由 Hibernate。为实体, <类>, <子类> 等使用。我们使用值类型 <属性>, <组件> 等,通常有一个 类型 属性。这个属性的值是 Hibernate的名字 映射类型。Hibernate 提供了一系列标准JDK值类型映射的 箱。你可以写你自己的映射类型和实现自己的自定义 转换策略。

除了集合,所有内置Hibernate类型 支持null语义。

5.2.2. 基本值类型

内置的 基本映射类型 可以 大致分为以下几点:

整形、长、短、浮子、双、性格、字节,布尔,是的没有,真的假的

从Java原语类型映射或包装器类 适当的(特定于供应商的)的SQL列类型。布尔,是的没有 和 真的假 都是替代编码 一个 Java 布尔 或 java朗布尔。

字符串

一个类型映射从 以 到 VARCHAR (或Oracle VARCHAR2)。

日期、时间、时间戳

类型映射从 java util日期 和 它的子类来SQL类型 日期,时间 和 时间戳 (或 等效)。

日历,日历上的日期

类型映射从 java util日历 SQL类型 时间戳 和 日期 (或同等)。

大十进制,大整数

类型映射从 java数学bigdecimal 和 java数学入biginteger 到 数字 (或Oracle 号码)。

语言环境、时区、货币

类型映射从 java util地区, java util时区 和 java util货币 到 VARCHAR (或Oracle VARCHAR2)。实例的 地区 和 货币 是 映射到他们的ISO代码。实例的 时区 映射到他们的 id。

类

一个类型映射从 java . lang . class 到 VARCHAR (或Oracle VARCHAR2)。一个 类 是 映射到它的完全限定名称。

二进制

映射到一个适当的字节数组SQL二进制类型。

文本

地图长字符串Java SQL 用LONGVARCHAR 或 文本 类型。

形象

地图长字节数组到SQL LONGVARBINARY。

可序列化的

可序列化的Java类型映射到一个适当的SQL二进制 类型。 您还可以显示Hibernate类型 可序列化的 品牌的可序列化的Java类或接口,不默认基本 类型。

clob、blob

类型映射为JDBC类 java sql clob 和 java sql blob 。 这些类型可以 不方便对于某些应用程序,因为blob或clob 对象不能被重用事务之外。 驱动程序支持 是不完整的和 不一致的。

物化clob

地图长字符串Java SQL CLOB 类型。 当读, CLOB 值是 物化成一个Java字符串立即。 一些司机 需要 CLOB 值是在阅读 一个事务。 一旦实现,Java字符串 可用外的事务。

物化blob

地图长Java字节数组到SQL BLOB 类型。 当读, BLOB 值是 立即物化成一个字节数组。 一些司机 需要 BLOB 值是在阅读 一个事务。 一旦物化,字节数组 可用外的事务。

imm日期,imm时间,imm时间戳,imm的日历, imm日历上的日期,imm serializable,imm二进制

类型映射为Java类型被认为是可变的。 这就是冬眠使某些优化适当 只有不变的Java类型,应用程序将 对象是不可变的。 例如,您不应该叫 日期凝固时间() 一个实例映射作为 imm时间戳 。 改变的价值 财产,已经改变了持久,应用程序 必须分配一个新的、不一致,对象属性。

独特的标识符的实体和集合可以是任何基础 类型除了 二进制, BLOB 和 CLOB 。 复合标识符也是允许的。 看到 下面更多的信息。

价值的基本类型有相应的 类型 常量定义在 org.hibernate.hibernate 。 对于 的例子,hibernate字符串 代表 字符串 类型。

5 2 3. 自定义值类型

这是相对容易的开发者创建自己的价值 类型。 例如,您可能想要持续性能的类型 java朗入biginteger 到 VARCHAR 列。 Hibernate不提供一个内置的类型对于这个。 定制 类型是不限于映射属性,或集合元素,来一个 单表列。 所以,例如,您可能有一个Java属性 getName() / setName() 类型的 以 这是持久化到列 first_name,初始,姓。

实现一个自定义类型,实现要么 org.hibernate.UserType 或 org.hibernate.CompositeUserType 和申报 属性使用完全限定类名的类型。 视图 org.hibernate.test.DoubleStringType 看样 事情是可能的。

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

注意使用 <列> 标记映射 属性到多个列。

这个 CompositeUserType , EnhancedUserType , UserCollectionType ,和 UserVersionType 接口提供支持更多 专业使用。

你甚至可以提供参数 userType 在 映射文件。 要做到这一点,你的 userType 必须 实现 org.hibernate.usertype.ParameterizedType 接口。 提供参数来定制类型,您可以使用 <类型> 元素在你的映射文件。

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

这个 userType 现在可以检索的值 参数命名 默认 从 属性 对象传递给它。

如果你经常使用某种 userType ,它是 有用的定义一个较短的名称。 您可以使用 < typedef > 元素。 typedef分配一个名字 自定义类型,也可以包含一个列表的默认参数值 类型参数化。

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

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

也可以覆盖参数配置 typedef逐个地通过使用类型参数的属性 映射。

尽管Hibernate的种类丰富的内置类型和支持 对于组件意味着你将很少需要使用一个自定义的类型,它是 考虑好实践使用自定义类型为非类 在应用程序中频繁出现。 例如,一个 MonetaryAmount 类是一个好的候选 CompositeUserType ,尽管它可能会映射作为一个组件。 这是抽象的一个原因。 用一个自定义的类型,你的映射文件可以预防方法的改变 货币值代表。

5.3. 映射一个类不止一次

它可以提供超过一个映射为一个特定的 持久化类。 在这种情况下,您必须指定一个 实体 名称 来消除歧义的实例之间两种映射实体。 默认情况下,实体名称是一样的类名。 Hibernate允许您指定实体名称在处理持久 对象,当编写查询,或当映射关联来命名的 实体。

```
<class name="Contract" table="Contracts"
```

```

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

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

```

协会现在被指定使用 实体名称 而不是 类。

注意

这个特性是不支持的注解

5.4. SQL引用标识符

你可以迫使Hibernate引用一个标识符在生成的SQL 通过将表或列名称映射中的引号 文档。 Hibernate将使用正确的报价方式为SQL 方言。 这通常是双引号,但SQL 服务器使用括号和MySQL使用引号。

```

@Entity @Table(name="`Line Item`")
class LineItem {
    @id @Column(name="`Item Id`") Integer id;
    @Column(name="`Item #`") int itemNumber
}

<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/> <generator class="assigned"/> </id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>

```

5.5. 生成的属性

生成的属性数值属性,生成的 由数据库。 通常,Hibernate应用程序需要 刷新 对象包含任何性质的 数据库正在生成值。 标记属性生成 然而,让应用程序可以将这个任务委托给冬眠。 当Hibernate问题一个SQL插入或更新一个实体,它 定义生成的属性,它立即发出一个选择之后来 检索生成的值。

属性标记为生成必须另外非可插入的 和非可更新。 只有 [版本](#), [时间戳](#),和 [简单属性](#),可以 标记为生成的。

从来没有 (默认):给定属性值 没有在数据库中生成。

插入 :给定属性值上生成 插入,但不是再生在后续更新。 属性像 创建日期属于这一类。 尽管 [版本](#) 和 [时间戳](#) 属性可以被 标记为生成的,该选项不可用。

总是 :属性值是生成的两对 插入和更新。

标记属性生成,使用 @ generated。

5.6. 柱变形金刚:读和写表达式

Hibernate允许您定制SQL它使用读和写 列的值映射到 [简单属性](#)。 对于 例如,如果您的数据库提供了一组数据加密功能,你可以调用它们个别列如下:

```

@Entity
class CreditCard {
    @Column(name="credit_card_num")
    @ColumnTransformer(
        read="decrypt(credit_card_num)",
        write="encrypt(?)")
    public String getCreditCardNumber() { return creditCardNumber; }
    public void setCreditCardNumber(String number) { this.creditCardNumber = number; }
    private String creditCardNumber;
}

```

或在XML

```

<property name="creditCardNumber">

```

```
<column
  name="credit_card_num"
  read="decrypt(credit_card_num)"
  write="encrypt(?)"/>
</property>
```

注意

你可以用复数形式 @ColumnTransformers 如果超过一列需要 定义其中任何一条规定。

如果一个属性使用更多的那一列,你必须使用 forColumn 属性来指定哪些列,表达式是目标。

```
@Entity
class User {
  @Type(type="com.acme.type.CreditCardType")
  @Columns( {
    @Column(name="credit_card_num"),
    @Column(name="exp_date") } )
  @ColumnTransformer(
    forColumn="credit_card_num",
    read="decrypt(credit_card_num)",
    write="encrypt(?)")
  public CreditCard getCreditCard() { return creditCard; }
  public void setCreditCard(CreditCard card) { this.creditCard = card; }
  private CreditCard creditCard;
}
```

Hibernate应用自定义表达式的时候自动的 属性查询中引用的。 这个功能类似于一个 派生属性 公式 有两个差异:

- » 房地产是由一个或多个列 出口作为自动模式生成。
- » 房地产是读写,而不是只读的。

这个 写 表达式,如果指定,必须包含 整整一个“ ? ”的占位符的值。

5.7. 辅助数据库对象

辅助数据库对象允许创建和删除的 任意的数据库对象。 结合Hibernate的模式 演化的工具,他们有能力完全定义一个用户模式 在Hibernate映射文件。 虽然专门为 创建和删除诸如触发器和存储过程,任何SQL 命令,可以运行通过 java sql语句执行() 方法是有效的(例,修改、插入等)。 主要有两种模式 定义辅助数据库对象:

第一种模式是显式地列出创建和删除命令 映射文件:

```
<hibernate-mapping>
...
<database-object>
  <create>CREATE TRIGGER my_trigger ...</create>
  <drop>DROP TRIGGER my_trigger</drop>
</database-object>
</hibernate-mapping>
```

第二个模式是提供一个自定义类,它的构造 创建和删除命令。 这个自定义类必须实现 org.hibernate.mapping.AuxiliaryDatabaseObject 接口。

```
<hibernate-mapping>
...
<database-object>
  <definition class="MyTriggerDefinition"/>
</database-object>
</hibernate-mapping>
```

此外,这些数据库对象可以选择范围如此 他们只适用于某些方言时使用。

```
<hibernate-mapping>
...
<database-object>
  <definition class="MyTriggerDefinition"/>
  <dialect-scope name="org.hibernate.dialect.Oracle9iDialect"/>
  <dialect-scope name="org.hibernate.dialect.Oracle10gDialect"/>
</database-object>
</hibernate-mapping>
```

注意

这个特性是不支持的注解

第六章。类型

表的内容

6.1. 值类型

但是。基本值类型

6.1.2. 复合类型

6.1.3. 集合类型

6.2. 实体类型

6.3. 类型分类的意义

6.4. 定制类型

6.4.1. 定制类型使用 `org.hibernate.type.Type`

6.4.2. 定制类型使用 `org.hibernate.usertype.UserType`

6.4.3. 定制类型使用 `org.hibernate.usertype.CompositeUserType`

6.5. 类型注册

作为一个对象/关系映射解决方案,Hibernate处理两个Java和JDBC的表示 应用程序数据。 一个在线目录的应用程序,例如,大多数可能已经 产品 对象具有许多属性如 sku , 名称 等等。对这些 个人属性,Hibernate必须能够阅读价值的数据库和写他们回来。 这 “整理” 的功能是一个 Hibernate类型,这是一个实现的 `org.hibernate.type.Type` 接口。此外,一个 Hibernate类型 描述行为的各个方面的Java类型,如 “如何 平等检查吗?” 或 “如何克隆?” 。值

重要

一个Hibernate类型既不是一个Java类型还是一个SQL数据类型,它提供了一个信息关于两。
当你遇到术语 类型 对于Hibernate来说要知道使用可能 参考Java类型,SQL / JDBC类型或
Hibernate类型。

Hibernate分类类型分为两个高级组:值类型(见 6.1节, “价值类型”)和 实体类型(见 6.2节, “实体类型”)。

6.1. 值类型

主要特色的一个值类型的事实是他们不定义他们自己的 生命周期。 我们说他们是 “属于” 别的东西(特别是一个实体,正如我们将看到) 它定义了他们的生命周期。 值类型是进一步分为3类:基本类型(见 但是部分, “基本值类型”)、复合类型(见 6.1.2节, “复合类型”) and集合类型(见 节6.1.3, “集合类型”)。

但是。基本值类型

规范为基本值类型是他们地图一个数据库值(列)到一个单一的, 非聚合的Java类型。 Hibernate提供了一些内置的基本类型,我们将礼物 在以下部分由Java类型。 主要是这些遵循自然映射的推荐 JDBC规范。 我们稍后会介绍如何重写这些映射和如何提供和使用 选择类型映射。

6.1.1.1以

`org.hibernate.type.StringType`

地图一个字符串到JDBC VARCHAR类型。 这是标准的映射为一个字符串 没有Hibernate指定类型。

注册在 字符串 和 以 在注册表类型(见 6.5节, “类型注册表”)。

`org.hibernate.type.MaterializedClob`

地图一个字符串到JDBC CLOB类型

注册在 物化clob 在注册表类型(见 6.5节, “类型注册表”)。

`org.hibernate.type.TextType`

地图一个字符串到JDBC用LONGVARCHAR类型

注册在 文本 在注册表类型(见 6.5节, “类型注册表”)。

6.1.1.2. java朗性格 (或char原始)

`org.hibernate.type.CharacterType`

地图一个char或 java朗性格 到一个JDBC CHAR

注册在 char 和 java朗性格 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 3. java朗布尔 (或布尔原始)

org.hibernate.type.BooleanType

地图一个布尔值,用于JDBC钻头类型

注册在 布尔 和 java朗布尔 在 注册表类型(见 6.5节,“类型注册表”)。

org.hibernate.type.NumericBooleanType

地图一个布尔值,用于JDBC整数类型作为0 = false,1 = true

注册在 数字逻辑 在注册表类型(见 6.5节,“类型注册表”)。

org.hibernate.type.YesNoType

地图一个布尔值,用于JDBC CHAR类型作为(' N ' | ' N ') = false,(' Y ' | ' Y ') = true

注册在 是的没有 在注册表类型(见 6.5节,“类型注册表”)。

org.hibernate.type.TrueFalseType

地图一个布尔值,用于JDBC CHAR类型作为(' F ' | ' F ') = false,(' T ' | ' T ') = true

注册在 真的假 在注册表类型(见 6.5节,“类型注册表”)。

6 1 1 4. java朗字节 (或字节原始)

org.hibernate.type.ByteType

地图一个字节或 java朗字节 到一个JDBC非常小的整数

注册在 字节 和 java朗字节 在 类型注册表(见 6.5节,“类型注册表”)。

6 1 1 5. java朗短 (或短原始)

org.hibernate.type.ShortType

地图短或 java朗短 到一个JDBC SMALLINT

注册在 短 和 java朗短 在 类型注册表(见 6.5节,“类型注册表”)。

6 1 1 6. java . lang . integer (或int原始)

org.hibernate.type.IntegerTypes

地图int或 java . lang . integer 到一个JDBC整数

注册在 int 和 java . lang . integer 在 类型注册表(见 6.5节,“类型注册表”)。

6 1 1 7. java朗长 (或长原始)

org.hibernate.type.LongType

地图一个或长或 java朗长 到一个JDBC长整型数字

注册在 长 和 java朗长 在 类型注册表(见 6.5节,“类型注册表”)。

6 1 1 8. java朗浮 (或浮原始)

org.hibernate.type.FloatType

地图一个浮动或 java朗浮 到一个JDBC浮

注册在 浮 和 java朗浮 在 类型注册表(见 6.5节,“类型注册表”)。

6 1 1 9. java朗双 (或双原始)

org.hibernate.type.DoubleType

地图或双 java朗双 到一个JDBC双

注册在 `双` 和 `java朗双` 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 10. java数学入biginteger

org.hibernate.type.BigIntegerType

地图一个 `java数学入biginteger` 到一个JDBC数字

注册在 `大整数` 和 `java数学入biginteger` 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 11. java数学bigdecimal

org.hibernate.type.BigDecimalType

地图一个 `java数学bigdecimal` 到一个JDBC数字

注册在 `大十进制` 和 `java数学bigdecimal` 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 12. java util日期 或 java sql时间戳

org.hibernate.type.TimestampType

地图一个 `java sql时间戳` 到一个JDBC时间戳

注册在 `时间戳`, `java sql时间戳` 和 `java util日期` 在注册表类型(见 6.5节, “类型注册表”)。

6 1 1 13. java sql时间

org.hibernate.type.TimeType

地图一个 `java sql时间` 到一个JDBC时间

注册在 `时间` 和 `java sql时间` 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 14. java sql日期

org.hibernate.type.DateType

地图一个 `java sql日期` 到一个JDBC日期

注册在 `日期` 和 `java sql日期` 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 15. java util日历

org.hibernate.type.CalendarType

地图一个 `java util日历` 到一个JDBC时间戳

注册在 `日历`, `java util日历` 和 `java.util.GregorianCalendar` 在注册表类型(见 6.5节, “类型注册表”)。

org.hibernate.type.CalendarDateType

地图一个 `java util日历` 到一个JDBC日期

注册在 `日历上的日期` 在注册表类型(见 6.5节, “类型注册表”)。

6 1 1 16. java util货币

org.hibernate.type.CurrencyType

地图一个 `java util货币` 到一个JDBC VARCHAR(使用货币代码)

注册在 `货币` 和 `java util货币` 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 17. java util地区

org.hibernate.type.LocaleType

地图一个 java util地区 到一个JDBC VARCHAR(使用区域代码)
注册在 地区 和 java util地区 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 18. java util时区

org.hibernate.type.TimeZoneType

地图一个 java util时区 到一个JDBC VARCHAR(使用时区ID)
注册在 时区 和 java util时区 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 19. java净url

org.hibernate.type.UrlType

地图一个 java净url 到一个JDBC VARCHAR(使用外部形式)
注册在 url 和 java净url 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 20. java . lang . class

org.hibernate.type.ClassType

地图一个 java . lang . class 到一个JDBC VARCHAR(使用类名)
注册在 类 和 java . lang . class 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 21. java sql blob

org.hibernate.type.BlobType

地图一个 java sql blob 到一个JDBC BLOB
注册在 BLOB 和 java sql blob 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 22. java sql clob

org.hibernate.type.ClobType

地图一个 java sql clob 到一个JDBC CLOB
注册在 CLOB 和 java sql clob 在 类型注册表(见 6.5节, “类型注册表”)。

6 1 1 23 byte[]

org.hibernate.type.BinaryType

地图一个原始的byte[]一个JDBC VARBINARY
注册在 二进制 和 byte[] 在 类型注册表(见 6.5节, “类型注册表”)。

org.hibernate.type.MaterializedBlobType

地图一个原始的byte[]一个JDBC BLOB
注册在 物化blob 在注册表类型(见 6.5节, “类型注册表”)。

org.hibernate.type.ImageType

地图一个原始的byte[]一个JDBC LONGVARBINARY
注册在 形象 在注册表类型(见 6.5节, “类型注册表”)。

6 1 1 24. Byte[]

org.hibernate.type.BinaryType

地图一个java . lang. Byte[]一个JDBC VARBINARY
注册在 包装二进制, byte[] 和 java朗byte[] 在注册表类型(见 6.5节, “类型注册表”)。

6.1.1.25 char[]

org.hibernate.type.CharArrayType

地图一个char[]一个JDBC VARCHAR

注册在 字符 和 char[] 在注册表类型(见 6.5节, “类型注册表”)。

6.1.1.26. 字符[]

org.hibernate.type.CharacterArrayType

地图一个java . lang. 字符[]一个JDBC VARCHAR

注册在 包装字符, 字符[] 和 java朗字符[] 在注册表类型(见 6.5节, “类型注册表”)。

6.1.1.27. java util uuid

org.hibernate.type.UUIDBinaryType

地图一个java跑龙套。 UUID来一个JDBC二进制

注册在 uuid二进制 和 java util uuid 在注册表类型(见 6.5节, “类型注册表”)。

org.hibernate.type.UUIDCharType

地图一个java跑龙套。 UUID来一个JDBC CHAR(虽然也很好对现有VARCHAR模式)

注册在 uuid char 在注册表类型(见 6.5节, “类型注册表”)。

org.hibernate.type.PostgresUUIDType

地图一个java跑龙套。 UUID到PostgreSQL UUID数据类型(通过 类型#其他 这是如何定义它的PostgreSQL JDBC驱动程序)。

注册在 pg uuid 在注册表类型(见 6.5节, “类型注册表”)。

6.1.1.28. io可序列化的java

org.hibernate.type.SerializableType

地图的java . lang实现者。 序列化到一个JDBC VARBINARY

不像其他的值类型,这种类型的多个实例。 它 得到注册一旦在 io可序列化的java 。 此外它得到注册在特定的io可序列化的java 实现类名称。

6.1.2. 复合类型

注意

Java持久化API调用这些嵌入的类型,而传统上称为他们冬眠 组件。 只是要注意,使用这两个术语,指的是同一件事的范围 讨论Hibernate。

组件的值代表聚合到一个单独的Java类型。 例如,您可能有一个地址类,骨料街道、城市、国家等信息或一个名字类 骨料的 部分一个人的名字。 在许多方面一个组件看起来完全就像一个实体。 他们 都是(一般来说)类写入针对应用程序。 他们都可能 引用其他特定于应用程序的类,以及集合和简单的JDK类型。 作为 前面所讨论的一样,唯一区别的是,一个工厂组件不拥有它自己的 生命周期也没有定义一个标识符。

6.1.3. 集合类型

重要

关键是理解,我们指的是集合本身,而不是它的内容。 集合的内容可以反过来是最基本的,组件 或实体类型(尽管不是 集合),但该集合本身是拥有。

集合都包含在 第七章, 集合映射。

6.2. 实体类型

实体的定义中会有详细论述 [第四章, 持久化类](#)。为这个讨论,这足以说实体(通常特定于应用程序的)类与表中的行。特别是他们相互关联行通过一个独特的标识符。由于这种独特的标识符,实体独立存在和定义自己的生命周期。作为一个例子,当我们删除一个会员,无论是用户和集团实体保持。

注意

这个概念可以修改的实体独立应用程序开发人员使用的概念小瀑布。小瀑布允许某些操作继续(或“级联”)在一个协会从一个实体到另一个。小瀑布都包含在细节 [第八章, 协会映射](#)。

6.3. 类型分类的意义

为什么我们花这么多时间分类各种类型的类型? 的意义是什么 区别吗?

主要的分类是实体类型和值类型之间。回顾我们说实体,通过自然他们独特的标识符,独立存在价值的其他对象而不。一个应用程序不能“删除”产品sku;相反,sku删除当产品本身删除(显然你可以更新sku的产品为null,让它“走开”,但即便在访问是通过产品)。

你也不能定义一个协会到该产品sku。你可以定义一个关联到产品基于它的sku,假设是独一无二的,但是,sku是完全不同的。

TBC.....

6.4. 定制类型

Hibernate使得它相对容易的开发者创建自己的价值类型。对于的例子,你可能想要持续性质的类型 java朗入biginteger 到 VARCHAR 列。自定义类型是不限于映射值到一个表列。因此,例如,你可能想要连接在一起 first_name, 初始和姓 列成一个以。

有三个方法来开发一个定制的Hibernate类型。作为一种手段,说明不同方法,让我们考虑这样一个用例,我们需要写一个 java数学bigdecimal 和 java util货币 成一个自定义 钱 类。

6.4.1. 定制类型使用 org.hibernate.type.Type

第一个方法是直接实现 org.hibernate.type.Type 接口(或者它的一个衍生品)。可能,你会更感兴趣,更具体 org.hibernate.type.BasicType 合同,将允许注册类型(见 [6.5节, “类型注册表”](#))。这么做的好处是,只要在注册元数据为一个特定的财产并不指定Hibernate类型使用,Hibernate将咨询注册为暴露属性类型。在我们的示例中,属性类型是 钱 ,这是关键,我们将使用我们的类型注册在注册表中:

例6.1. 定义和注册自定义类型

```
public class MoneyType implements BasicType {
    public String[] getRegistrationKeys() {
        return new String[] { Money.class.getName() };
    }

    public int[] sqlTypes(Mapping mapping) {
        // We will simply use delegation to the standard basic types for BigDecimal and Currency for many
        // Type methods...
        return new int[] {
            BigDecimalType.INSTANCE.sqlType(),
            CurrencyType.INSTANCE.sqlType(),
        };
        // we could also have honored any registry overrides via...
        //return new int[] {
        //    mappings.getTypeResolver().basic( BigDecimal.class.getName() ).sqlTypes( mappings )[0],
        //    mappings.getTypeResolver().basic( Currency.class.getName() ).sqlTypes( mappings )[0]
        //};
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throw
    assert names.length == 2;
    BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
```

```

Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
return amount == null && currency == null
    ? null
    : new Money( amount, currency );
}

public void nullSafeSet(PreparedStatement st, Object value, int index, boolean[] settable, SessionImplementor s,
    throws SQLException {
    if ( value == null ) {
        BigDecimalType.INSTANCE.set( st, null, index );
        CurrencyType.INSTANCE.set( st, null, index+1 );
    }
    else {
        final Money money = (Money) value;
        BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
        CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
    }
}

...
}

Configuration cfg = new Configuration();
cfg.registerTypeOverride( new MoneyType() );
cfg...;

```

重要

它是重要的,我们注册的类型 之前 添加映射。

6.4.2. 定制类型使用 org.hibernate.usertype.UserType

注意

两 `org.hibernate.usertype.UserType` 和 `org.hibernate.usertype.CompositeUserType` 最初是 添加到孤立用户代码从内部改变 `org.hibernate.usertype.UserType` 接口。

第二种方法是使用 `org.hibernate.usertype.UserType` 接口,它提供了一个有点的简化视图 `org.hibernate.usertype.CompositeUserType` 接口。使用 `org.hibernate.usertype.UserType` ,我们的 钱 自定义类型将如下所示:

例6.2. 定义自定义UserType

```

public class MoneyType implements UserType {
    public int[] sqlTypes() {
        return new int[] {
            BigDecimalType.INSTANCE.sqlType(),
            CurrencyType.INSTANCE.sqlType(),
        };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, Object owner) throws SQLException {
        assert names.length == 2;
        BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
            ? null
            : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index) throws SQLException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}

```

```
}
```

没有多大区别 org.hibernate.type.BasicType 的例子和 org.hibernate.usertype.UserType 的例子,但这只是因为 所示的代码片段。如果你选择 org.hibernate.usertype.UserType 方法 有几个方法你将需要实现比 org.hibernate.usertype.UserType 。

6.4.3. 定制类型使用 org.hibernate.usertype.CompositeUserType

第三个和最后的方法是使用 org.hibernate.usertype.CompositeUserType 接口,它不同于 org.hibernate.usertype.UserType 在这 使我们能够提供的信息来处理Hibernate的成分在 钱 类(特别是2属性)。这将给我们的能力,例如,引用 金额 属性在HQL查询。使用 org.hibernate.usertype.CompositeUserType ,我们的 钱 自定义类型将如下所示:

例6.3. 定义自定义CompositeUserType

```
public class MoneyType implements CompositeUserType {
    public String[] getPropertyNames() {
        // ORDER IS IMPORTANT! it must match the order the columns are defined in the property mapping
        return new String[] { "amount", "currency" };
    }

    public Type[] getPropertyTypes() {
        return new Type[] { BigDecimalType.INSTANCE, CurrencyType.INSTANCE };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object getPropertyValue(Object component, int propertyIndex) {
        if ( component == null ) {
            return null;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                return money.getAmount();
            }
            case 1: {
                return money.getCurrency();
            }
            default: {
                throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
    }

    public void setPropertyValue(Object component, int propertyIndex, Object value) throws HibernateException {
        if ( component == null ) {
            return;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                money.setAmount( (BigDecimal) value );
                break;
            }
            case 1: {
                money.setCurrency( (Currency) value );
                break;
            }
            default: {
                throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
    }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws HibernateException {
        assert names.length == 2;
        BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
            ? null
            : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, SessionImplementor session) throws HibernateException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;

```



```
        BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
        CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
    }
}
...
}
```

6.5. 类型注册

Hibernate使用注册中心内部的基本类型(见 [但是部分](#), “基本值类型”)当 它需要解决具体的 org.hibernate.type类型 使用在 某些 情况。 它还提供了一种方法,应用程序添加额外的基本类型注册以及 覆盖标准的基本类型注册。

注册一个新类型或覆盖现有类型登记,应用程序将利用 registerTypeOverride 方法 org.hibernate.cfg配置 类当引导 Hibernate。 例如,假设你想要使用你的自定义,冬眠 SuperDuperStringType ;在引导你会叫:

例6.4. 覆盖标准 StringType

```
Configuration cfg = ...;
cfg.registerTypeOverride( new SuperDuperStringType() );
```

的参数 registerTypeOverride 是 org.hibernate.type.BasicType 这是一个专业化的吗 org.hibernate.type类型 我们之前看到的。 它 添加一个方法:

例6.5. 从BasicType.java片段

```
/**
 * Get the names under which this type should be registered in the type registry.
 *
 * @return The keys under which to register this type.
 */
public String[] getRegistrationKeys();
```

一种方法是使用继承(SuperDuperStringType 延伸 org.hibernate.type.StringType);另一个是使用代表团。

第7章。 集合映射

表的内容

7.1. 持久化集合

7.2. 如何映射集合

7.2.1. 收集外键

7.2.2. 索引集合

7.2.3. 集合的基本类型和可嵌入的对象

7.3. 先进的集映射

7.3.1. 排序的集合

7.3.2. 双向关联

7.3.3. 双向联想索引集合

7.3.4. 三元关联

7.3.5. 使用一个< idbag >

7.4. 收集的例子

7.1. 持久化集合

Hibernate也允许坚持自然集合。 这些 持久的集合可以包含任何其他Hibernate类型, 包括:基本类型、自定义类型、组件和引用其他 实体。 价值之间的区别和引用语义是在这 上下文很重要。 一个对象在一个集合可能处理 “价值” 语义(其生命周期完全取决于集所有者), 或者它可能是一个引用到另一个实体有自己的生命周期。 在 后者的情况下,只有 “链接” 两物体之间被认为 是一个国家持有的集合。

作为一个要求持久字段必须收藏价值 声明为一个接口类型(见 例7.2, “集合映射使用onetomany和@JoinColumn”)。实际的接口 可能 java util集, java util收集, java util列表, java util地图, java util sortedset, java.util.SortedMap 或任何你喜欢的(“任何你 像” 意味着你将不得不写的一个实现 org.hibernate.usertype.UserCollectionType)。

请注意,在 例7.2, “集合映射使用onetomany和@JoinColumn” 实例变量 部分 被初始化的一个实例吗 HashSet 。这是最好的方式来初始化集合 价值属性的新实例化(非持久性)实例。 当 你让实例持久,通过调用 persist() , Hibernate会取代 HashSet 与一个 实例的Hibernate自己的实现 集。 是 意识到以下错误:

例7.1. Hibernate使用自己的集合实现

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);

kittens = cat.getKittens(); // Okay, kittens collection is a Set
(HashSet) cat.getKittens(); // Error!
```

通过Hibernate持久性集合注入像 HashMap, HashSet, TreeMap, TreeSet 或 arraylist,根据不同的接口类型。

有通常的行为集合实例的值类型。 他们 自动保存一个持久化对象引用的时候,是吗 自动删除当未引用。 如果一个集合也离了一个持久化对象到另一个,它的元素可能会移动从一个 表到另一个。 两个实体不能共享相同的引用 收集实例。 由于底层的关系模型,收藏价值属性不支持null值语义。 Hibernate不区分一个空引用和一个集合 空集合。

注意

使用持久集合相同的方式使用普通Java 集合。 然而,确保你理解的语义 双向关联(见 节7 3 2, “双向关联”)。

7.2. 如何映射集合

使用注释可以映射 收集 年代, 列表 年代, 地图 年代和 集 年代的相关实体使用onetomany和 @ManyToMany。 集合的一个基本的或可嵌入的类型使用 @ElementCollection。 在最简单的情况下一个集合映射看起来像 这个:

例7.2. 集合映射使用,@JoinColumn onetomany

```
@Entity
public class Product {

    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();

    @Id
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinColumn(name="PART_ID")
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}

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

产品描述一个单向关系部分使用 加入柱部分id。 在这种单向一到许多场景中你可以 也使用一个连接表见 例7.3, “集合映射使用onetomany和@JoinTable” 。

例7.3. 集合映射使用,@JoinTable onetomany

```
@Entity
public class Product {

    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();
```

```

@Id
public String getSerialNumber() { return serialNumber; }
void setSerialNumber(String sn) { serialNumber = sn; }

@OneToMany
@JoinTable(
    name="PRODUCT_PARTS",
    joinColumns = @JoinColumn( name="PRODUCT_ID"),
    inverseJoinColumns = @JoinColumn( name="PART_ID")
)
public Set<Part> getParts() { return parts; }
void setParts(Set parts) { this.parts = parts; }
}

@Entity
public class Part {
    ...
}

```

没有描述任何物理映射(没有 @JoinColumn 或 @JoinTable), 一个单向一到许多与加入表使用。表名是 连接表名称的所有者, 另一边的表 的名字。外键名称(s)引用所有者表是 连接表的所有者, 业主主键列(年代) 的名字。外键名称(s)引用的另一边是 连接的业主财产名称, 另一边的主 键列(s)的名字。一个独特的约束添加到外键 引用对方表来反映一个很多。

让看看现在如何使用Hibernate映射集合 映射文件。在这种情况下,第一步是选择正确的映射 元素。这取决于类型的接口。例如,一个 <设置> 元素是用于映射属性的 类型 集。

例7.4. 映射一个集使用<设置>

```

<class name="Product">
  <id name="serialNumber" column="productSerialNumber"/>
  <set name="parts">
    <key column="productSerialNumber" not-null="true"/>
    <one-to-many class="Part"/>
  </set>
</class>

```

在 例7.4, “映射使用<设置>一组” 一个一对多关联 连接 产品 和 部分 实体。这 协会要求存在外键列和可能的一个 索引列 的部分 表。这种映射失去 某些语义正常的Java集合:

- » 的一个实例,包含实体类不能属于更多 的实例的集合。
- » 的一个实例,包含实体类不能出现在更多的 比一个值集合的索引。

再进一步的使用 <一对多> 标签我们看到它有以下选项。

例7.5. 选择<一对多>元素

```

<one-to-many
  class="ClassName"
  not-found="ignore|exception"
  entity-name="EntityName"
  node="element-name"
  embed-xml="true|false"
/>

```



类 (需要)的名字 相关的类。



没有找到 (可选,默认 异常):指定缓存如何标识符 这将处理引用缺失行。忽略 会对一个失踪的行作为一个空吗 协会。



实体名称 (可选):实体名称 关联的类,来代替 类。

这个 <一对多> 元素不需要 声明任何列。也不需要指定 表 任何地方的名字。

警告















如果外键列的一个 <一对多> 协会宣布 非空 ,你必须声明 <键> 映射过的非null = " true " 或使用双向 协会 与集合映射标记 逆= " true "。看到 节7 3 2, “双向关联”。

除了 <设置> 标签所示的 例7.4, “映射使用<设置>一组”,也有 <列表>, <地图>, <袋>, <数组> 和 <原始数组> 映射元素。这个 <地图> 元素是代表:

例7.6. <地图>元素的映射

```
<map
  name="propertyName"
  table="table_name"
  schema="schema_name"
  lazy="true|extra|false"
  inverse="true|false"
  cascade="all|none|save-update|delete|all-delete-orphan|delete-orphan"
  sort="unsorted|natural|comparatorClass"
  order-by="column_name asc|desc"
  where="arbitrary sql where condition"
  fetch="join|select|subselect"
  batch-size="N"
  access="field|property|ClassName"
  optimistic-lock="true|false"
  mutable="true|false"
  node="element-name|."
  embed-xml="true|false"
>

  <key .... />
  <map-key .... />
  <element .... />
</map>
```

-  名称 :集合属性名称
-  表 (可选,默认属性 名称):收集表的名称。 它不用于 一对多关联。
-  模式 (可选):一个表的名称 模式覆盖模式上声明根元素
-  懒惰 (可选,默认 真正的):禁用惰性抓取并指定 这个协会总是急切地获取。 它也可以 用于启用 “额外的懒惰” 抓取 大多数操作不会 初始化集合。 这是适合大 集合。
-  逆 (可选,默认 假):是这个集合的 “逆” 双向关联的结束。
-  级联 (可选,默认 没有):使操作级联到孩子 实体。
-  排序 (可选):指定一个排序 收集与 自然 或一个给定的排序顺序 比较器类。
-  类型的排序 (可选):指定一个表 列或列定义的迭代顺序 地图,集 或袋,一起 用一个可选 asc 或 Desc 。
-  在 (可选):指定一个任意的 SQL 在 条件时使用 检索或删除集合。 这是有用的如果 收集需要包含只有一个子集的 可用 数据。
-  取 (可选,默认 选择):选择外连接抓取之间, 抓取的顺序选择,抓取的顺序 subselect。
-  批量大小 (可选,默认 1):指定一个 “批量大小” 懒洋洋地 获取实例的集合。
-  访问 (可选,默认 财产):战略Hibernate使用 访问集合属性值。
-  乐观锁 (可选,默认 真正的):指定状态的变化 在增量收集结果的拥有实体的 版本。 的一对多关联,你可能想要禁用这 个 设置。
-  可变 (可选,默认 真正的):一个值 假 指定集合的元素永远不会改变。 这 允许轻微的性能优化在某些情况下。

在探究了基本映射集合在前面 现在我们将重点段落细节像物理映射的考虑, 索引集合和集合的值类型。

7.2.1. 收集外键

在数据库级收集的实例 外键的实体,拥有集。 这个外键 被称为 收集关键列 ,或 列表的集合。 集合映射的键列 这个 @JoinColumn 注释分别 <键> XML元素。

可以有一个可为空特性约束的外键列。 对于大多数集合,这是暗示。 对于单向一对多 协会、外键列可以为空默认情况下,所以 你可能 需要指定

```
@JoinColumn(nullable=false)
```

或

```
<key column="productSerialNumber" not-null="true"/>
```

外键约束可以使用 在删除 级联 。 在XML这一过程可以表达通过:

```
<key column="productSerialNumber" on-delete="cascade"/>
```

在注释Hibernate注释@OnDelete具体必须 被使用。

```
@OnDelete(action=OnDeleteAction.CASCADE)
```

看到 部分5 1 11 3, “关键” 为更多的信息 关于 <键> 元素。

7 2 2. 索引集合

在接下来的段落我们参观一下索引 集合 列表 和 地图 如何在他们的指数可以映射在冬眠。

7.2.2.1. 列表

列表可以映射在两种不同的方式:

- 是有序列表的顺序不是体现在 数据库
- 作为索引列表,订单是物化的 数据库

订购列表在内存中,添加 `@javax.persistence.OrderBy` 你的财产。 这 注释需要作为参数列表的属性(逗号分隔的 目标实体)和订单相应的集合(例如 `firstname asc,年龄desc`),如果字符串是空的, 收集将命令通过主键的目标 实体。

例7.7. 有序列表使用 `@OrderBy`

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

    @OneToMany(mappedBy="customer")
    @OrderBy("number")
    public List<Order> getOrders() { return orders; }
    public void setOrders(List<Order> orders) { this.orders = orders; }
    private List<Order> orders;
}

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

    public String getNumber() { return number; }
    public void setNumber(String number) { this.number = number; }
    private String number;

    @ManyToOne
    public Customer getCustomer() { return customer; }
    public void setCustomer(Customer customer) { this.customer = customer; }
    private Customer customer;
}

-- Table schema
|-----| |-----|
| Order  | | Customer |
|-----| |-----|
| id      | | id      |
| number  | |         |
| customer_id | |         |
|-----| |-----|
```

来存储索引值在一个专用的列,可以使用 `@javax.persistence.OrderColumn` 注释 你的财产。 这个注释描述列名和 属性的列保持索引值。 这列是 托管在表包含有关联的外键。 如果 列名称没有指定,默认的名称 引用属性,其次是强调,紧随其后的是秩序 (下面的例子,这将是 订单订单)。

例7.8. 显式索引列使用 `@OrderColumn`

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

    @OneToMany(mappedBy="customer")
    @OrderColumn(name="orders_index")
    public List<Order> getOrders() { return orders; }
    public void setOrders(List<Order> orders) { this.orders = orders; }
    private List<Order> orders;
}

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

    public String getNumber() { return number; }
    public void setNumber(String number) { this.number = number; }
    private String number;
}
```

```

@ManyToOne
public Customer getCustomer() { return customer; }
public void setCustomer(Customer customer) { this.customer = customer; }
private Customer number;
}

-- Table schema
|-----| |-----|
| Order  | | Customer |
|-----| |-----|
| id      | | id      |
| number  | |-----|
| customer_id |
| orders_order |
|-----|

```

注意

我们建议你保留 `@org.hibernate.annotations.IndexColumn` 用法来 `@OrderColumn` 除非你是利用基础属性。这个 `base` 属性允许您定义索引值的第一个元素(即按基础指数)。通常的值是 0 或 1。默认是 0 像在 Java。



再次查看Hibernate映射文件等效, 索引数组或列表总是的类型 `Integer` 和映射使用 `<列表索引>` 元素。映射的列包含连续整数, 屈指可数的, 默认为零。

例7.9. 索引列表元素在xml索引的集合 映射

```

<list-index
  column="column_name"
  base="0|1|.." />

```

-  列表 (需要) 的名称 列控股集索引值。
-  基地 (可选, 默认 0): 索引列的值, 对应于列表的第一个元素或数组。

如果你的表没有索引列, 你还希望使用 `列表` 作为属性类型, 您可以绘制 `财产` 作为一个Hibernate `<袋>`。一个包是否没有保持其秩序当它从数据库检索, 但它可以选择进行排序或命令。

7.2.2.2. 地图

这个问题 `地图` 年代是关键 值存储。有几个选项。地图可以借他们的钥匙 从一个关联的实体属性或有专门的列 存储一个显式的关键。

使用一个目标实体的财产作为关键的地图, 使用 `@MapKey(name = "myProperty")`, 在那里 `myProperty` 是一个属性的名字在目标实体。当使用 `@MapKey` 没有名字 attribute, 目标实体主键使用。地图键使用相同的列 属性指出。没有额外的列定义 持有地图的关键, 因为地图键代表一个目标属性。是 意识到一旦加载, 关键是不再保持同步 财产。换句话说, 如果你改变属性值, 关键 不会改变自动地在Java模型。

例7.10. 使用目标实体属性图主要通过 @MapKey

```

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

    @OneToMany(mappedBy="customer")
    @MapKey(name="number")
    public Map<String, Order> getOrders() { return orders; }
    public void setOrders(Map<String, Order> order) { this.orders = orders; }
    private Map<String, Order> orders;
}

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

    public String getNumber() { return number; }
    public void setNumber(String number) { this.number = number; }
    private String number;

    @ManyToOne

```

```

public Customer getCustomer() { return customer; }
public void setCustomer(Customer customer) { this.customer = customer; }
private Customer number;
}

-- Table schema
|-----| |-----|
| Order  | | Customer |
|-----| |-----|
| id     | | id     |
| number | |-----|
| customer_id |
|-----|

```

另外地图键映射到一个专用的列或列。为了定制映射使用下列之一注释:

- ▶ `@MapKeyColumn` 如果地图键是一个基本类型。如果你不指定列的名字,这个名字的其次是强调紧随其后的财产关键 是使用(例如 订单关键)。
- ▶ `@MapKeyEnumerated` / `@MapKeyTemporal` 如果地图键类型 分别枚举或 日期。
- ▶ `@MapKeyJoinColumn` / `@MapKeyJoinColumns` 如果地图键类型是另一个实体。
- ▶ `@AttributeOverride` / `@attributeoverrides` 当映射键是一个嵌入的对象。使用 关键。 作为一个前缀可嵌入的对象。你 属性名。

您还可以使用 `@MapKeyClass` 定义 键的类型,如果你不使用泛型。

例7.11. 地图的关键,基本类型使用 `@MapKeyColumn`

```

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

    @OneToMany @JoinTable(name="Cust_Order")
    @MapKeyColumn(name="orders_number")
    public Map<String,Order> getOrders() { return orders; }
    public void setOrders(Map<String,Order> orders) { this.orders = orders; }
    private Map<String,Order> orders;
}

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

    public String getNumber() { return number; }
    public void setNumber(String number) { this.number = number; }
    private String number;

    @ManyToOne
    public Customer getCustomer() { return customer; }
    public void setCustomer(Customer customer) { this.customer = customer; }
    private Customer number;
}

-- Table schema
|-----| |-----| |-----|
| Order  | | Customer | | Cust_Order |
|-----| |-----| |-----|
| id     | | id     | | customer_id |
| number | |-----| | order_id   |
| customer_id | | orders_number |
|-----| |-----|

```

注意

我们建议您从迁移 `@org.hibernate.annotations.MapKey` /
`@org.hibernate.annotation.MapKeyManyToOne` 到 新标准在前面描述的方法

使用Hibernate映射文件存在等价的概念 种的注释。你必须使用 `<地图键>` , `<图关键许许多多>` 和 `<复合映射键>` 。
`<地图键>` 用于任何基本类型, `<图关键许许多多>` 为一个实体 参考和 `<复合映射键>` 对于一个 复合类型。




例7.12. 地图关键xml映射元素

```

<map-key
    column="column_name"




```

```
formula="any SQL expression"
type="type_name"
node="@attribute-name"
length="N"/>
```

-  列 (可选)的名字 列控股集索引值。
-  公式 (可选):一个SQL公式 用来评估关键的地图。
-  类型 (必需):地图的类型 键。

例7.13地图键多对多

```
<map-key-many-to-many
  column="column_name"
  formula="any SQL expression"
  class="ClassName"
/>
```

-  列 (可选)的名字 外键列的索引值集合。
-  公式 (可选):一个平方公式 评估的外键映射的关键。
-  类 (必需):实体类 作为地图的关键。

7.2.3. 集合的基本类型和可嵌入的对象

在某些情况下你不需要联系两个实体但 简单地创建一个收集的基本类型或可嵌入的对象。使用 `@ElementCollection` 对于这种情况。

例7.14. 基本类型映射的集合通过 `@ElementCollection`

```
@Entity
public class User {
    [...]
    public String getLastName() { ...}

    @ElementCollection
    @CollectionTable(name="Nicknames", joinColumns=@JoinColumn(name="user_id"))
    @Column(name="nickname")
    public Set<String> getNicknames() { ... }
}
```

收集表控股集的数据的设置使用 `@CollectionTable` 注释。如果省略了 收集表名称默认为连接的名称 包含实体和集合的名称属性,隔开 一个下划线。在我们的例子中,它将是 用户昵称。

列控股基本类型设置使用 `@column` 注释。如果省略,列名 默认为属性名:在我们的例子中,它将是 昵称。

但是你不局限于基本类型、集合类型可以 任何可嵌入的对象。 覆盖的列可嵌入的对象 在收集表,使用 `@AttributeOverride` 注释。

例7.15. `@ElementCollection`为嵌入对象

```
@Entity
public class User {
    [...]
    public String getLastName() { ...}

    @ElementCollection
    @CollectionTable(name="Addresses", joinColumns=@JoinColumn(name="user_id"))
    @AttributeOverrides({
        @AttributeOverride(name="street1", column=@Column(name="fld_street"))
    })
    public Set<Address> getAddresses() { ... }
}

@Embeddable
public class Address {
    public String getStreet1() {...}
    [...]
}
```


这样的可嵌入的对象不能包含一个集合 本身。

注意

在 `@AttributeOverride` ,你必须使用 值。 前缀覆盖的属性 可嵌入的对象用于映射值和 关键。 前缀覆盖的属性 可嵌入的对象用于地图的关键。

```
@Entity
public class User {
    @ElementCollection
    @AttributeOverrides({
        @AttributeOverride(name="key.street1", column=@Column(name="fld_street")),
        @AttributeOverride(name="value.stars", column=@Column(name="fld_note"))
    })
    public Map<Address,Rating> getFavHomes() { ... }
```




注意

我们建议您从迁移 `@org.hibernate.annotations.CollectionOfElements` 新 `@ElementCollection` 注释。

使用映射文件的方法映射值集合 使用 `<元素>` 标签。 例如:

例7.16. `<元素>` 标记集合值使用映射 文件

```
<element
  column="column_name"
  formula="any SQL expression"
  type="typename"
  length="L"
  precision="P"
  scale="S"
  not-null="true|false"
  unique="true|false"
  node="element-name"
/>
```

-  列 (可选)的名字 列控股集元素值。
-  公式 (可选):一个SQL公式 评估元素。
-  类型 (必需):该类型的 集合的元素。

7.3. 先进的集映射

7.3.1. 排序的集合

Hibernate支持集合实现 `java.util.SortedMap` 和 `java.util.SortedSet` 。 与注释你声明一个 排序比较器使用 `@Sort` 。 你选择之间的 比较器类型未排序的,自然或自定义。 如果你想用你的 自己的比较器实现,您还必须指定 实现类使用 比较器 属性。 注意,您需要使用 `SortedSet` 或 `SortedMap` 接口。

例7.17. 分类收集与`@Sort`

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Sort(type = SortType.COMPARATOR, comparator = TicketComparator.class)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

使用Hibernate映射文件你指定一个比较器的 映射文件与 `<分类>` :

例7.18. 分类收集使用xml映射

```
<set name="aliases"
```

```

        table="person_aliases"
        sort="natural">
        <key column="person"/>
        <element column="name" type="string"/>
    </set>

    <map name="holidays" sort="my.custom.HolidayComparator">
        <key column="year_id"/>
        <map-key column="hol_name" type="string"/>
        <element column="hol_date" type="date"/>
    </map>

```

允许的值的 排序 属性是 未分类的，自然 叫 一个类实现 java实效比较器。

提示

实际上像集合进行排序 java util treeset 或 java util treemap 。

如果你想订购的数据库本身集合元素，使用 类型的排序 属性的 集 ， 袋 或 地图 映射。这个解决方案 实现使用 LinkedHashMap 或 LinkedHashMap 和执行命令的SQL 查询和不在内存。

例7.19. 排序在数据库使用类型的排序

```

<set name="aliases" table="person_aliases" order-by="lower(name) asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>

```

注意

的价值 类型的排序 属性是一个SQL 排序,而不是一个HQL订购。

联想甚至可以在运行时按任意的标准 使用一个集合 filter() :

例7.20. 通过查询筛选排序

```
sortedUsers = s.createFilter( group getUsers(), "order by this.name" ).list();
```

7 3 2. 双向关联

一个 双向关联 允许导航 来自 “结束” 的协会。两种双向 协会的支持:

一对多

设置或袋一端价值和单值的 其他

多对多

设置或袋两端价值

通常存在许多一个协会的所有者 一边的双向关系。相应的人很多 协会是在这种情况下的注释 onetomany(mappedBy = ...)

例7.21. 双向一对多和多对一方作为协会 所有者

```

@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
        ...
    }
}

@Entity
public class Soldier {

```

```

@ManyToOne
@JoinColumn(name="troop_fk")
public Troop getTroop() {
    ...
}

```

队伍 有一个双向一对多吗 关系 士兵 通过 队伍 财产。 你不必(不能)定义 任何物理映射 mappedBy 侧。

映射一个双向一对多,一对多的一侧 拥有侧,你必须删除 mappedBy 元素和设置多个对一个 @JoinColumn 作为 可插入的和 可更新为false。 这个解决方案不优化 将会产生额外的更新语句。

例7.22. 双向关联有1至许多一侧 所有者

```

@Entity
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") //we need to duplicate the physical information
    public Set<Soldier> getSoldiers() {
        ...
    }

    @Entity
    public class Soldier {
        @ManyToOne
        @JoinColumn(name="troop_fk", insertable=false, updatable=false)
        public Troop getTroop() {
            ...
        }
    }
}

```

mapping如何的双向映射看起来像吗 Hibernate映射xml ? 你定义一个双向一对多 协会由一个一对多关联映射到相同的 表(s)作为一种多对一的协会和宣告结束多值 逆= " true "。

例7.23. 双向一对多通过Hibernate映射文件

```

<class name="Parent">
  <id name="id" column="parent_id"/>
  ....
  <set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
  </set>
</class>

<class name="Child">
  <id name="id" column="child_id"/>
  ....
  <many-to-one name="parent"
    class="Parent"
    column="parent_id"
    not-null="true"/>
</class>

```

映射一个协会的一端与 逆= " true " 不影响操作的小瀑布,因为这些是正交的概念。

定义一个多对多关联逻辑上使用 @ManyToMany 注释。 你也有描述 关联表和加入条件使用 @JoinTable 注释。 如果协会 是双向,一边是老板和一方是 逆结束(ie. 它将被忽略当更新的关系 值在协会表):

例7.24. 通过@ManyToMany多对多关联

```

@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns=@JoinColumn(name="EMPER_ID"),
        inverseJoinColumns=@JoinColumn(name="EMPEE_ID")
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}

```

```

@Entity
public class Employee implements Serializable {
    @ManyToOne(
        cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        mappedBy = "employees",
        targetEntity = Employer.class
    )
    public Collection getEmployers() {
        return employers;
    }
}

```

在这个例子 @JoinTable 定义了一个名称, 一组加入列, 一组 反向连接列。后者的是列的协会 表指的 员工 主键 (“另一边”)。像前面所看到的, 对方不需要 (不能) 描述: 一个简单的物理映射 mappedBy 参数, 该参数包含所有者侧财产 把这两个名字。

像任何其他注解, 最值在一个许多人猜测 许多关系。没有描述任何物理映射在一个 单向多对多以下规则应用。表名 是老板的 并置表名, _ 和 另一边表名。外键名称(s) 引用所有者 表的并置业主表名称, _ 和 业主主键列(年代)。外键名称(s) 引用 另一边是串联的主人属性名, _ , 另一边的主键列(年代)。这些都是 同样的规则用于单向一对多 关系。

例7.25. 默认值 @ManyToOne (单向)

```

@Entity
public class Store {
    @ManyToOne(cascade = CascadeType.PERSIST)
    public Set<City> getImplantedIn() {
        ...
    }
}

@Entity
public class City {
    ... //no bidirectional relationship
}

```

一个 存储城市 用作连接表。这个 Store_id 列是一个外国的钥匙 商店 表。这个 implantedIn_id 列是一个外国的钥匙 城市 表。

没有描述任何物理映射在一个双向的许多人 许多以下规则应用。表名是串联的 表名称的所有者, _ 和对方表名。外键名称 (s) 引用所有者表连接 另一边的属性名, _ 和老板 主键列(年代)。外键名称(s) 引用其他 一边是串联的主人属性名, _ , 另一边的主键列(年代)。这些都是 同样的规则用于单向一对多 关系。

例7.26. 默认值 @ManyToMany (双向)

```

@Entity
public class Store {
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    public Set<Customer> getCustomers() {
        ...
    }
}

@Entity
public class Customer {
    @ManyToMany(mappedBy = "customers")
    public Set<Store> getStores() {
        ...
    }
}

```

一个 存储客户 用作连接表。这个 商店id 列是一个外国的钥匙 商店 表。这个 客户id 列是一个外国的钥匙 客户 表。

使用Hibernate映射文件你可以映射一个双向 多对多关联映射两个多对多关联到 相同的数据库表, 并宣布为一端 逆。

注意

你不能选择一个索引的集合。

例7.27, “许多许多协会使用Hibernate映射文件” 显示了一个 双向多对多的关联, 说明了每个 类别可以有多个项目, 每个项目可以在许多 类别:

例7.27. 多对多关联使用Hibernate映射文件

```

<class name="Category">
  <id name="id" column="CATEGORY_ID"/>
  ...
  <bag name="items" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
  </bag>
</class>

<class name="Item">
  <id name="id" column="ITEM_ID"/>
  ...

  <!-- inverse end -->
  <bag name="categories" table="CATEGORY_ITEM" inverse="true">
    <key column="ITEM_ID"/>
    <many-to-many class="Category" column="CATEGORY_ID"/>
  </bag>
</class>

```

只更改到逆结束的协会 不 坚持。 这意味着Hibernate有两个 在内存中的表示每个双向关联:一个链接 从A到B和另一个链接从 B A。 这是更容易理解 如果你认为Java对象模型和多多对多 在Javais创建的关系:

例7.28. 效应的逆与非逆侧的许许多多 协会

```

category.getItems().add(item);      // The category now "knows" about the relationship
item.getCategories().add(category);  // The item now "knows" about the relationship

session.persist(item);               // The relationship won't be saved!
session.persist(category);           // The relationship will be saved

```

这个非逆侧是用来保存内存中表示 到数据库。

7 3 3. 双向联想索引集合

有一些额外的考虑双向 映射与索引集合(一端被表示为一个 <列表> 或 <地图>)当 使用Hibernate映射文件。 如果有一个房 地产的子类 映射到索引列可以使用 逆= " true " 在集合映射:

例7.29. 双向关联与索引收集

```

<class name="Parent">
  <id name="id" column="parent_id"/>
  ....
  <map name="children" inverse="true">
    <key column="parent_id"/>
    <map-key column="name"
      type="string"/>
    <one-to-many class="Child"/>
  </map>
</class>

<class name="Child">
  <id name="id" column="child_id"/>
  ....
  <property name="name"
    not-null="true"/>
  <many-to-one name="parent"
    class="Parent"
    column="parent_id"
    not-null="true"/>
</class>

```

如果没有这样的财产在子类,协会 不能被认为是真正双向。 那是,有信息 可用的一端协会,所不具备的 另一端。 在这种情况下,您不能映射集合 逆= " true "。 相反,您可以使用以下 映射:

例7.30. 双向关联与索引收集,但没有索引 列

```

<class name="Parent">
  <id name="id" column="parent_id"/>
  ....
  <map name="children">
    <key column="parent_id"

```

```

        not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
</class>

```

注意,在这种映射,收藏价值的结束 协会负责更新外键。

7 3 4. 三元关联

有三种可能的方法来映射一个三元 协会。一种方法是使用一个 地图 与一个 协会作为其指数:

例7.31. 三元关联映射

```

@Entity
public class Company {
    @Id
    int id;
    ...
    @OneToMany // unidirectional
    @MapKeyJoinColumn(name="employee_id")
    Map<Employee, Contract> contracts;
}

// or

<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>

```

第二种方法是改变协会作为一个实体 类。这是最常见的方法。最后一个选择是使用 复合元素,这将在稍后讨论。

7 3 5. 使用一个 < idbag >

大多数的多对多关联和收藏 先前所显示的值映射到表中所有与复合键,甚至 尽管有人建议,实体应该合成 标识符(代理键)。一个纯协会表似乎并没有 从代理键中获益很多,虽然一套组合 值 可能。因为这个原因Hibernate提供了 功能,让你多对多关联映射和集合 一个表的值与一个代理键。

这个 < idbag > 元素允许您将一个 列表 (或 收集)与包 语义。例如:

```

<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>

```

一个 < idbag > 有一个合成id发生器,就像一个实体类。不同的代理键分配给每个 收集行。Hibernate不,然而,提供任何机制 发现代理键值的特定行。

更新的性能 < idbag > 取代常规 <袋>。Hibernate可以 个别行有效地定位和更新或删除它们 单独,类似于一个列表,地图 或设置。

在当前实现中,原生 标识符生成策略不支持 < idbag > 标识符的集合。

7.4. 收集的例子

本节涵盖了收集的例子。

下面的类有一个收集的 孩子 实例:

例7.32. 示例类 父 和 孩子

```
public class Parent {
    private long id;
    private Set<Child> children;

    // getter/setter
    ...
}

public class Child {
    private long id;
    private String name

    // getter/setter
    ...
}
```

如果每个孩子最多,一位家长,最自然的映射 一对多关联:

例7.33. 一个到多个单向 亲子 关系使用注释

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany
    private Set<Child> children;

    // getter/setter
    ...
}

public class Child {
    @Id
    @GeneratedValue
    private long id;
    private String name;

    // getter/setter
    ...
}
```

例7.34. 一个到多个单向 亲子 关系使用映射文件

```
<hibernate-mapping>

  <class name="Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children">
      <key column="parent_id"/>
      <one-to-many class="Child"/>
    </set>
  </class>

  <class name="Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
  </class>

</hibernate-mapping>
```

这个映射到下面的表定义:

例7.35. 表定义为单向 父 —— 孩子 关系

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

如果父母是 需要 ,使用一个双向 一对多关联:

例7.36. 一个到多个双向 亲子 关系使用注释

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(mappedBy="parent")
    private Set<Child> children;

    // getter/setter
    ...
}

public class Child {
    @Id
    @GeneratedValue
    private long id;

    private String name;

    @ManyToOne
    private Parent parent;

    // getter/setter
    ...
}
```

例7.37. 一个到多个双向 亲子 关系使用映射文件

```
<hibernate-mapping>
  <class name="Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children" inverse="true">
      <key column="parent_id"/>
      <one-to-many class="Child"/>
    </set>
  </class>

  <class name="Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
    <many-to-one name="parent" class="Parent" column="parent_id" not-null="true"/>
  </class>
</hibernate-mapping>
```

注意 非空 约束:

例7.38. 表定义为双向 父 —— 孩子 关系

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                    primary key,
                    name varchar(255),
                    parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

另外,如果这个协会必须单向你可以 执行 非空 约束。

例7.39. 执行非空约束在单向关系使用 注释

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(optional=false)
    private Set<Child> children;

    // getter/setter
    ...
}

public class Child {
    @Id
    @GeneratedValue
    private long id;
    private String name;

    // getter/setter
    ...
}
```

例7.40. 执行非空约束在单向关系使用 映射文件

```
<hibernate-mapping>

  <class name="Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children">
      <key column="parent_id" not-null="true"/>
      <one-to-many class="Child"/>
    </set>
  </class>

  <class name="Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
  </class>

</hibernate-mapping>
```

另一方面,如果一个孩子有多个父母,多对多 协会是合适的。

例7.41. 许许多多 亲子关系 使用注释

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @ManyToMany
    private Set<Child> children;

    // getter/setter
    ...
}

public class Child {
    @Id
    @GeneratedValue
    private long id;

    private String name;

    // getter/setter
    ...
}
```

例7.42. 许许多多 父子 关系 使用映射文件

```
<hibernate-mapping>

  <class name="Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children" table="childset">
      <key column="parent_id"/>
      <many-to-many class="Child" column="child_id"/>
    </set>
  </class>

  <class name="Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
  </class>

</hibernate-mapping>
```

表定义:

例7.43. 表定义为许许多多relationship

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

更多的例子和一个完整的解释,一个父/子 关系映射,看到 [23章, 例如:父/子](#) 更多 信息。更复杂的关联映射的覆盖 下一章。

第八章. 协会映射

表的内容

8.1. 介绍

8.2. 单向关联

8.2.1. 多对一

8.2.2开始. 一对一

8 2 3. 一对多

8.3. 单向关联与联接表

夹带了本条件8.3.1. 一对多

8.3.2. 多对一

8 3 3. 一对一

8 3 4. 多对多

8.4. 双向关联

8 4 1. 一对多或多对一的

8 4 2. 一对一

8.5. 双向联想联接表

8.5.1皆是如此. 一对多或多对一的

8.5.2. 一个对一个

8 5 3. 多对多

8.6. 更复杂的关联映射

8.1. 介绍

协会映射通常是最困难的事正确实现。在 本节我们检查一些规范的情况下,启动 与单向映射,然后双向病例。我们将使用 人和 地址 总共 这个例子。

协会将分类多样性和他们是否映射到一个介入的 加入表。

Nullable外键不认为是良好的实践在传统数据 造型,所以我们的示例不使用可空外键。这不是一个 要求的Hibernate、映射将工作如果你放弃 可为空特性约束。

8.2. 单向关联

8.2.1. 多对一

一个 单向多对一关联 是最 常见的一种单向协会。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    not-null="true"/>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

8.2.2开始。一对一的

一个 单向一对一关联在一个外键 几乎是相同的。唯一的区别是列唯一约束。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    unique="true"
    not-null="true"/>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

一个 单向一对一关联在一个主键 通常使用一个特殊的id发生器在这个例子中,然而,我们已经扭转了方向 协会:

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
</class>

<class name="Address">
  <id name="id" column="personId">
    <generator class="foreign">
      <param name="property">person</param>
    </generator>
  </id>
  <one-to-one name="person" constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

8 2 3. 一对多

一个 单向一对多关联在一个外键 是一个不寻常的情况下,不推荐。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses">
    <key column="personId"
      not-null="true"/>
    <one-to-many class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, personId bigint not null )
```

你应该使用一个连接表这样的协会。

8.3. 单向关联与联接表

夹带了本条件8.3.1. 一对多

一个 单向一对多关联在一个连接表 是更好的选择。 指定 独特的= " true " , 改变了从多对多,一对多的多重性。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      unique="true"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

8.3.2. 多对一

一个 单向多对一关联在一个连接表 是常见的在协会是可选的。 例如:

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <join table="PersonAddress"
    optional="true">
    <key column="personId" unique="true"/>
    <many-to-one name="address"
      column="addressId"
      not-null="true"/>
  </join>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null )
```

```
create table Address ( addressId bigint not null primary key )
```

8 3 3. 一对一的

一个 单向一对一关联在一个连接表 是可能的, 但是非常不寻常的。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <join table="PersonAddress"
    optional="true">
    <key column="personId"
      unique="true"/>
    <many-to-one name="address"
      column="addressId"
      not-null="true"
      unique="true"/>
  </join>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

8 3 4. 多对多

最后,下面是一个示例 单向多对多的关联。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

8.4. 双向关联

8 4 1. 一对多或多对一的

一个 双向多对一关联 是最常见的类型的协会。 下面的例子说明了标准的父/子 关系。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    not-null="true"/>
</class>
```

```

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
  <set name="people" inverse="true">
    <key column="addressId"/>
    <one-to-many class="Person"/>
  </set>
</class>

```

```

create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )

```

如果你使用一个列表或其他索引收集, 设置 关键 列的外键 非空。Hibernate将管理协会从集合侧来维护索引 的每个元素, 使对方几乎逆通过设置 更新= " false " 和 插入= " false " :

```

<class name="Person">
  <id name="id"/>
  ...
  <many-to-one name="address"
    column="addressId"
    not-null="true"
    insert="false"
    update="false"/>
</class>

<class name="Address">
  <id name="id"/>
  ...
  <list name="people">
    <key column="addressId" not-null="true"/>
    <list-index column="peopleIdx"/>
    <one-to-many class="Person"/>
  </list>
</class>

```

如果底层的外键列 非空 ,它 重要的是,你定义的呢 过的非Null = " true " 在 <键> 元素集的映射。 不仅 申报 过的非Null = " true " 在一个可能的嵌套 <列> 元素,但在 <键> 元素。

8 4 2. 一对一的

一个 双向一对一关联在一个外键 是常见的:

```

<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <many-to-one name="address"
    column="addressId"
    unique="true"
    not-null="true"/>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
  <one-to-one name="person"
    property-ref="address"/>
</class>

```

```

create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )

```

一个 双向一对一关联在一个主键 使用特殊的id发生器:

```

<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <one-to-one name="address"/>
</class>

<class name="Address">
  <id name="id" column="personId">
    <generator class="foreign">
      <param name="property">person</param>
    </generator>
  </id>
  <one-to-one name="person"
    constrained="true"/>
</class>

```

```

create table Person ( personId bigint not null primary key )

```

```
create table Address ( personId bigint not null primary key )
```

8.5. 双向联想联接表

8.5.1 皆是如此。 一对多或多对一的

下面是一个示例 双向一对多关联在一个联接表。 这个 逆= " true " 可以去的一端 协会,收集,或加入。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses"
    table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      unique="true"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
  <join table="PersonAddress"
    inverse="true"
    optional="true">
    <key column="addressId"/>
    <many-to-one name="person"
      column="personId"
      not-null="true"/>
  </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

8.5.2. 一个对一个

一个 双向一对一关联在一个联接表 是可能的, 但是非常不寻常的。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <join table="PersonAddress"
    optional="true">
    <key column="personId"
      unique="true"/>
    <many-to-one name="address"
      column="addressId"
      not-null="true"
      unique="true"/>
  </join>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
  <join table="PersonAddress"
    optional="true"
    inverse="true">
    <key column="addressId"
      unique="true"/>
    <many-to-one name="person"
      column="personId"
      not-null="true"
      unique="true"/>
  </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

8 5 3. 多对多

下面是一个示例 双向多对多的关联。

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
  <set name="people" inverse="true" table="PersonAddress">
    <key column="addressId"/>
    <many-to-many column="personId"
      class="Person"/>
  </set>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

8.6. 更复杂的关联映射

更复杂的协会join 极其 罕见的。 Hibernate处理更复杂的情况下使用 SQL碎片嵌入映射文档。 例如,如果一个表 账户信息与历史数据定义 accountNumber , effectiveEndDate 和 effectiveStartDate 列,它将被映射,如下所示:

```
<properties name="currentAccountKey">
  <property name="accountNumber" type="string" not-null="true"/>
  <property name="currentAccount" type="boolean">
    <formula>case when effectiveEndDate is null then 1 else 0 end</formula>
  </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

然后您可以映射的一个协会 电流 实例, 一个以null effectiveEndDate ,通过使用:

```
<many-to-one name="currentAccountInfo"
  property-ref="currentAccountKey"
  class="AccountInfo">
  <column name="accountNumber"/>
  <formula>'1'</formula>
</many-to-one>
```

在一个更复杂的示例,假设之间的关系 员工 和 组织 维护 在一个 就业 表充满历史的就业数据。 一个协会会员的 最近 雇主,最近的 startDate可以,可能会映射在以下方式:

```
<join>
  <key column="employeeId"/>
  <subselect>
    select employeeId, orgId
    from Employments
    group by orgId
    having startDate = max(startDate)
  </subselect>
  <many-to-one name="mostRecentEmployer"
    class="Organization"
    column="orgId"/>
</join>
```

此功能允许一定程度的创造性和灵活性,但它更实用 来处理这些类型的情况下使用HQL或标准查询。

第9章. 组件映射

表的内容

- 9.1. 依赖对象
- 9.2. 依赖对象的集合
- 9.3. 组件图指数
- 9.4. 组件标识符作为复合
- 9.5. 动态组件

的概念 组件 是在几个不同的上下文中重用和目的 在冬眠。

9.1. 依赖对象

一个组件是一个包含对象保存为一个值类型而不是一个实体 参考。 “组件”指的是面向对象的概念组成 而不是利用组件。例如,你可以一个人这样的:模型

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    }
    .....
    }
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
    }
}
```

现在 名称 可以被保存为一个组件的 人 。 名称 定义了getter 和setter方法其持久的属性,但它不需要申报 任何接口或标识符属性。

我们的Hibernate映射应该像这样:

```
<class name="eg.Person" table="person">
  <id name="Key" column="pid" type="string">
    <generator class="uuid"/>
  </id>
  <property name="birthday" type="date"/>
  <component name="Name" class="eg.Name"> <!-- class attribute optional -->
    <property name="initial"/>
    <property name="first"/>
    <property name="last"/>
  </component>
</class>
```

人表会列 PID, 生日, 初始, 第一 和 最后。

像值类型,组件不支持共享引用。 换句话说,两个 人 可以有相同的名字,但这两人对象将包含两个独立的 名称的对象,只有 “同一” 的价值。 null值语义的一个组件 特设。 当重新加载包含对象,Hibernate将承担 ,如果所有组件列是空的,然后整个组件是 null。 这是适合大多数场合。

一个组件的属性可以是任何类型(收藏,多对一的冬眠 协会、其他组件,等等)。 嵌套的组件应该 不 被视为一种异国情调的用

法。Hibernate是为了支持细粒度 对象模型。

这个 <组件> 元素允许 <父> 子元素的一个属性映射组件类作为参考回 包含的实体。

```
<class name="eg.Person" table="person">
  <id name="Key" column="pid" type="string">
    <generator class="uuid"/>
  </id>
  <property name="birthday" type="date"/>
  <component name="Name" class="eg.Name" unique="true">
    <parent name="namedPerson"/> <!-- reference back to the Person -->
    <property name="initial"/>
    <property name="first"/>
    <property name="last"/>
  </component>
</class>
```

9.2. 依赖对象的集合

组件的集合(例如支持类型的数组 名称)。 声明你的组件的集合 取代 <元素> 标记与 <复合元素> 标签:

```
<set name="someNames" table="some_names" lazy="true">
  <key column="id"/>
  <composite-element class="eg.Name"> <!-- class attribute required -->
    <property name="initial"/>
    <property name="first"/>
    <property name="last"/>
  </composite-element>
</set>
```

重要

如果你定义一个 集 复合元素,它是 重要实现 equals() 和 hashCode() 正确。

复合元素可以包含组件而不是集合。 如果你 复合元素包含 组件,使用 <嵌套复合元素> 标签。 这种情况下是一个集合的组件构成 自己有组件。 你可能想要考虑如果 一对多关联是更适当的。 重塑的 复合元素作为一个实体,但是请注意,即使Java模型是相同的,关系模型和持久性语义仍 略有不同。

一个复合元素映射不支持null能力属性 如果您正在使用一个 <设置>。 没有单独的主键列 在复合元素表。 Hibernate 使用每个列的值记录的标识删除对象时, 这是不可能与空值。 你必须要么只使用 过的非null属性在一个复合元素或选择一个 <列表>, <地图>, <袋> 或 < idbag >。

一种特殊情况一个复合元素是一个复合元素包含一个嵌套的 <多对一的> 元素。 这种映射允许 你额外的多对多映射表的列的 协会 复合元素类。 以下是一个多对多的关联 从 秩序 到 项 ,在那里 purchaseDate, 价格 和 数量 是属性的关联:

```
<class name="eg.Order" .... >
  ....
  <set name="purchasedItems" table="purchase_items" lazy="true">
    <key column="order_id">
      <composite-element class="eg.Purchase">
        <property name="purchaseDate"/>
        <property name="price"/>
        <property name="quantity"/>
        <many-to-one name="item" class="eg.Item"/> <!-- class attribute is optional -->
      </composite-element>
    </set>
  </class>
```

不能有一个参考购买对方一边的 双向关联导航。 组件是值类型和 不允许共享引用。 一个 购买 可以在 组一个 秩序 ,但它不能被引用的 项 在同一时间。

甚至三元(或第四纪等)协会是可能的:

```
<class name="eg.Order" .... >
  ....
  <set name="purchasedItems" table="purchase_items" lazy="true">
    <key column="order_id">
      <composite-element class="eg.OrderLine">
        <many-to-one name="purchaseDetails" class="eg.Purchase"/>
        <many-to-one name="item" class="eg.Item"/>
      </composite-element>
    </set>
  </class>
```

复合元素可以出现在查询中使用相同的语法 联想到其他实体。

9.3. 组件图指数

这个 <复合映射> 元素允许您映射 组件类的关键 地图。 确保你覆盖 hashCode() 和 equals() 上正确地 组件类。

9.4. 组件标识符作为复合

您可以使用一个组件作为一个标识符的一个实体类。你的组件类必须满足一定的要求:

- ▶ 它必须实现 io 可序列化的 java 。
- ▶ 它必须重新实现 equals() 和 hashCode() 始终按照数据库的 组合键的概念平等。

注意

在Hibernate3,虽然第二需求不是一个绝对的硬 Hibernate的要求,建议。

你不能用一个 IdentifierGenerator 生成复合键。相反,应用程序必须指定自己的标识符。

使用 <复合id > 标签,嵌套 <关键属性> 元素,代替通常的 < Id > 宣言。例如, OrderLine 类有一个主键,取决于 (复合)主键 秩序。

```
<class name="OrderLine">
  <composite-id name="id" class="OrderLineId">
    <key-property name="lineId"/>
    <key-property name="orderId"/>
    <key-property name="customerId"/>
  </composite-id>
  <property name="name"/>
  <many-to-one name="order" class="Order"
    insert="false" update="false">
    <column name="orderId"/>
    <column name="customerId"/>
  </many-to-one>
  ....
</class>
```

任何外键引用 OrderLine 表现在 复合。在你宣布此为其它类的映射。一个协会 到 OrderLine 映射:

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
  <column name="lineId"/>
  <column name="orderId"/>
  <column name="customerId"/>
</many-to-one>
```

提示

这个 列 元素是一个替代 列 属性无处不在。使用 列 元素只给了更多的声明 选项,其中大部分是有用的在利用 hbm2ddl

一个 多对多 协会 OrderLine 也 使用复合外键:

```
<set name="undeliveredOrderLines">
  <key column name="warehouseId"/>
  <many-to-many class="OrderLine">
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
  </many-to-many>
</set>
```

收集 OrderLine 年代在 秩序 将 使用:

```
<set name="orderLines" inverse="true">
  <key>
    <column name="orderId"/>
    <column name="customerId"/>
  </key>
  <one-to-many class="OrderLine"/>
</set>
```

这个 <一对多> 元素声明没有列。

如果 OrderLine 本身拥有一个集合,它也有一个组合 外键。

```
<class name="OrderLine">
  ....
  ....
  <list name="deliveryAttempts">
    <key> <!-- a collection inherits the composite key type -->
      <column name="lineId"/>
```

```

        <column name="orderId"/>
        <column name="customerId"/>
    </key>
    <list-index column="attemptId" base="1"/>
    <composite-element class="DeliveryAttempt">
        ...
    </composite-element>
</set>
</class>

```

9.5. 动态组件

你也可以映射的属性类型 地图：

```

<dynamic-component name="userAttributes">
    <property name="foo" column="FOO" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component>

```

的语义 <动态组件> 映射是相同的 到 <组件> 。 的优势,这种映射 能够确定的实际属性bean在部署时只是 通过编辑映射文档。 运行时的操作映射文档 也有可能,使用DOM解析器。 你也可以访问和改变,冬眠的 元模型配置时间通过 配置 对象。

第十章。 继承映射

表的内容

10.1. 这三种策略

- 给。 表每个类层次
- 10 1 2. 表每个子类
- 10 1 3. 表每个子类:使用一个鉴别器
- 10 1 4. 每个类层次混合表与表每个子类
- 10 1 5. 每个具体类一张表
- 10 1 6. 表使用隐式多态实现每个具体类
- 10 1 7. 混合隐多态性与其他继承映射

10.2. 限制

10.1. 这三种策略

Hibernate支持三种基本继承映射策略:

- 表每个类层次
- 表每个子类
- 每个具体类一张表

此外,Hibernate支持第四,稍微不同的种 多态:

- 隐式多态

可以使用不同的映射策略不同 分支机构相同的继承层次结构。 然后您可以使用隐式 多态性实现多态性在整个层次结构。 然而, Hibernate不支持混合 <子类>, <加入子类> 和 <联盟子类> 映射在同样的根 <类> 元素。 可以混合在一起 每个层次的表和表每个子类策略根据 同样的 <类> 元素,结合 <子类> 和 <加入> 元素(见下面的例子)。

可以定义 子类, 联盟子类, 和 加入子类 映射在单独的映射文档下方 hibernate映射 。 这允许您扩展一个类层次结构通过添加 一个新的映射文件。 您必须指定一个 延伸 属性在子类映射, 命名一个以前映射超类。 以前这个特性使排序映射 文件很重要。 因为Hibernate3,映射文件的排序是无关紧要的使用 扩展关键字。 排序在一个映射文件仍然需要被定义为超类 在子类。

```

<hibernate-mapping>
    <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
        <property name="name" type="string"/>
    </subclass>
</hibernate-mapping>

```

给。 表每个类层次

假设我们有一个接口 付款 与实现者 CreditCardPayment, CashPayment, 和 ChequePayment 。 每个层次的映射表 显示在以下方式:

```

<class name="Payment" table="PAYMENT">

```

```

<id name="id" type="long" column="PAYMENT_ID">
  <generator class="native"/>
</id>
<discriminator column="PAYMENT_TYPE" type="string"/>
<property name="amount" column="AMOUNT"/>
...
<subclass name="CreditCardPayment" discriminator-value="CREDIT">
  <property name="creditCardType" column="CCTYPE"/>
  ...
</subclass>
<subclass name="CashPayment" discriminator-value="CASH">
  ...
</subclass>
<subclass name="ChequePayment" discriminator-value="CHEQUE">
  ...
</subclass>
</class>

```

正好有一个表是必需的。有一个限制这种映射策略:列声明的子类,如 CCTYPE,不能有非空约束。

10 1 2. 表每个子类

一个表每个子类映射看起来像这样:

```

<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="native"/>
  </id>
  <property name="amount" column="AMOUNT"/>
  ...
  <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
    <key column="PAYMENT_ID"/>
    <property name="creditCardType" column="CCTYPE"/>
    ...
  </joined-subclass>
  <joined-subclass name="CashPayment" table="CASH_PAYMENT">
    <key column="PAYMENT_ID"/>
    ...
  </joined-subclass>
  <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
    <key column="PAYMENT_ID"/>
    ...
  </joined-subclass>
</class>

```

四个表是必需的。三个子类的表有主键,键协会超类表所以关系模型实际上是一个一对一的关联。

10 1 3. 表每个子类:使用一个鉴别器

Hibernate的表每个子类不需要鉴别器列。其他对象/关系映射器使用不同的实现,需要每个子类表类型鉴别器列在超类表。采用的方法 Hibernate是更加难以实现,但可以说更多正确的从关系的观点。如果你想使用一个鉴别器列与表的每个子类的策略,你可以结合使用 <子类> 和 <加入>,如下所示:

```

<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="native"/>
  </id>
  <discriminator column="PAYMENT_TYPE" type="string"/>
  <property name="amount" column="AMOUNT"/>
  ...
  <subclass name="CreditCardPayment" discriminator-value="CREDIT">
    <join table="CREDIT_PAYMENT">
      <key column="PAYMENT_ID"/>
      <property name="creditCardType" column="CCTYPE"/>
      ...
    </join>
  </subclass>
  <subclass name="CashPayment" discriminator-value="CASH">
    <join table="CASH_PAYMENT">
      <key column="PAYMENT_ID"/>
      ...
    </join>
  </subclass>
  <subclass name="ChequePayment" discriminator-value="CHEQUE">
    <join table="CHEQUE_PAYMENT" fetch="select">
      <key column="PAYMENT_ID"/>
      ...
    </join>
  </subclass>
</class>

```

可选的 fetch = "选择" 声明告诉Hibernate 不要拿 ChequePayment 子类数据使用一个外连接查询时的超类。

10 1 4. 每个类层次混合表与表每个子类

你甚至可以把表每层次和表每个子类的策略 使用以下的方法:

```
<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="native"/>
  </id>
  <discriminator column="PAYMENT_TYPE" type="string"/>
  <property name="amount" column="AMOUNT"/>
  ...
  <subclass name="CreditCardPayment" discriminator-value="CREDIT">
    <join table="CREDIT_PAYMENT">
      <property name="creditCardType" column="CCTYPE"/>
    </join>
  </subclass>
  <subclass name="CashPayment" discriminator-value="CASH">
    ...
  </subclass>
  <subclass name="ChequePayment" discriminator-value="CHEQUE">
    ...
  </subclass>
</class>
```

对于任何这些映射策略,一个多态关联到根 付款 类映射使用 <多对一的>。

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

10 1 5. 每个具体类一张表

有两种方法我们可以映射到每个具体类一张表 策略。首先,您可以使用 <联盟子类>。

```
<class name="Payment">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="sequence"/>
  </id>
  <property name="amount" column="AMOUNT"/>
  ...
  <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
    <property name="creditCardType" column="CCTYPE"/>
  </union-subclass>
  <union-subclass name="CashPayment" table="CASH_PAYMENT">
    ...
  </union-subclass>
  <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
    ...
  </union-subclass>
</class>
```

涉及三个表为子类。每个表定义列 类的所有属性,包括继承的属性。

这种方法的局限性是,如果一个属性被映射的 超类,列名必须是相同的所有子类的表。 身份发生器战略联盟是不允许子类继承。主键种子必须在所有子类共享联合 一个层次结构。

如果你的超类是抽象的,地图与 文摘 = " true "。如果它不是抽象的,额外的表(默认 付款 在上面的示例中),是需要保存的实例的超类。

10 1 6. 表使用隐式多态实现每个具体类

另一种方法是使用隐式多态:

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
  <id name="id" type="long" column="CREDIT_PAYMENT_ID">
    <generator class="native"/>
  </id>
  <property name="amount" column="CREDIT_AMOUNT"/>
  ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
  <id name="id" type="long" column="CASH_PAYMENT_ID">
    <generator class="native"/>
  </id>
  <property name="amount" column="CASH_AMOUNT"/>
  ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
```

```

<id name="id" type="long" column="CHEQUE_PAYMENT_ID">
  <generator class="native"/>
</id>
<property name="amount" column="CHEQUE_AMOUNT"/>
...
</class>

```

注意, 付款 接口 是没有提及。 还要注意,属性的 付款 是 在每个子类映射。 如果你想避免重复,考虑 使用XML实体 (例如, [<! 实体allproperties系统" allproperties. xml" >] 在 doctype 声明和 % allproperties; 在映射)。

这种方式的缺点是Hibernate不生成SQL 联盟 年代当执行多态查询。

这种映射的策略,一个多态的协会 付款 通常映射使用 <任何> 。

```

<any name="payment" meta-type="string" id-type="long">
  <meta-value value="CREDIT" class="CreditCardPayment"/>
  <meta-value value="CASH" class="CashPayment"/>
  <meta-value value="CHEQUE" class="ChequePayment"/>
  <column name="PAYMENT_CLASS"/>
  <column name="PAYMENT_ID"/>
</any>

```

10 1 7. 混合隐多态性与其他继承映射

因为子类 是每个映射在他们自己的 <类> 元素,和自 付款 仅仅是一个接口),每个子类可以吗 很容易被另一个继承层次结构的一部分。 你仍然可以使用多态 查询 付款 接口。

```

<class name="CreditCardPayment" table="CREDIT_PAYMENT">
  <id name="id" type="long" column="CREDIT_PAYMENT_ID">
    <generator class="native"/>
  </id>
  <discriminator column="CREDIT_CARD" type="string"/>
  <property name="amount" column="CREDIT_AMOUNT"/>
  ...
  <subclass name="MasterCardPayment" discriminator-value="MDC"/>
  <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
  <id name="id" type="long" column="TXN_ID">
    <generator class="native"/>
  </id>
  ...
  <joined-subclass name="CashPayment" table="CASH_PAYMENT">
    <key column="PAYMENT_ID"/>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
  </joined-subclass>
  <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
    <key column="PAYMENT_ID"/>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
  </joined-subclass>
</class>

```

再一次, 付款 是没有提及。 如果我们 执行一个查询 付款 接口, 例子 从付款 ,休眠 自动返回的实例 CreditCardPayment (和它的子类,因为他们也实现 付款), CashPayment 和 ChequePayment ,但 没有实例的 NonelectronicTransaction 。

10.2. 限制

有限度的“隐式多态”的方法来 每个具体类的表映射策略。 有那么 限制性的局限性 <联盟子类> 映射。

下面的表显示了限制的表/具体类 映射和隐式多态实现的,在冬眠。

表10.1. 特性的继承映射

继 承 策 略	多态多对一	多态一对一	多态的一对多	多态多对多	多态 load()/ get()	多态查 询	多态
	<多对一的>	<一对一>	<一对多>	<多对多>	年代得到(付款. 类.id)	从付款p	从订
表/ 类 层 次 结 构 表 每							

个 子 类 每 个 具 体 类 表 (联 盟 子 类) 每 个 具 体 类 一 张 表 (隐 式 多 态)	<多对一的>	<一对一>	<一对多>	<多对多>	年代得到(付款。类,id)	从付款p 从订
	<多对一的>	<一对一>	<一对多> (逆= " true " 唯一的)	<多对多>	年代得到(付款。类,id)	从付款p 从订
	<任何>	不支持	不支持	<许多任何>	s.createCriteria(付款类)。添加(Restrictions.idEq(id)).uniqueResult()	从付款p 不支

第11章。 处理的对象

表的内容

11.1. Hibernate对象状态

11.2. 使对象持久

11.3. 加载一个对象

11.4. 查询

11.4.1. 执行查询

11.4.2. 过滤收集

11.4.3. 标准查询

11.4.4. 原生SQL查询

11.5. 修改持久化对象

11.6. 修改分离对象

11.7. 自动状态检测

11.8. 删除持久对象

11.9. 在两个不同的数据存储复制对象

11.10. 冲洗会话

11.11. 过渡持久性

11.12. 使用元数据

Hibernate是一个完整的对象/关系映射解决方案,不仅 盾牌开发者从底层数据库的详细信息管理 系统,但也提供了 状态管理 的对象。这是,与SQL的管理 语句 在 常见的JDBC / SQL持久化层,一个自然的面向对象的观点 在Java应用程序的持久性。

换句话说,Hibernate应用程序开发人员应该总是认为 关于 状态 他们的对象,不一定 关于执行的SQL语句。 这部分是照顾着 Hibernate和是唯一相关的应用程序开发人员在调整 系统的性能。

11.1. Hibernate对象状态

Hibernate定义和支持以下对象状态:

- » 瞬态 —— 一个对象是瞬态如果它 刚刚被实例化使用吗 新 运营商, 和它没有关联的冬眠 会话 。 它没有持久表示在数据库中并没有标识符 值已被分配。 瞬态实例将被摧毁 垃圾收集器如果应用不持有一个参考 了。使用Hibernate 会话 做一个对象持久性(和让Hibernate照顾这个SQL语句 需要执行这个转换)。
- » 持久 —— 一个持久实例有一个 表示在数据库和一个标识符值。 它可能只是 已经保存或装载,然而,它是根据定义的范围的 会话 。 Hibernate会发现任何变化 对一个对象在持久状态和同步的国家 数据库在该工作单元完成。 开发者不 执行手册 更新 语句,或 删除 语句应该当一个对象 瞬态。
- » 分离 —— 一个超然的实例是一个对象 那一直存在,但它的 会话 一直 关闭。 这个对象的引用仍然是有效的,当然,和 分离实例甚至可能被修改在这个状态。 一个分离 实例可以连接一个新的 会话 在一个 后来时间点,使它(和所有的修改)持久性再一次。 这个特性允许一个编程模型对长期运行的单位 的工作,需要用户思考时间。 我们称他们 应用程序事务 ,即, ,一个工作单元 从用户的角度。

现在我们将讨论美国和状态转换(和 Hibernate触发的方法更详细的转换)。

11.2. 使对象持久

新实例化一个持久化类的实例被认为是瞬态 通过Hibernate。我们可以做一个瞬态实例持久 通过将方法与一个会话:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex("M");
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

如果猫已生成的标识符,该标识符生成并分配给猫 当 save() 被称为。如果猫有分配标识符,或一个复合键,该标识符应分配给猫实例之前,把 save()。您还可以使用 persist() 而不是 save(),语义定义的 EJB3早期草案。

- » persist() 使瞬态实例持久。然而,它并不保证标识符值 将分配给持久实例立即,作业可能发生在冲洗时间。persist() 也可以保证它不会执行插入 声明如果外部调用事务的边界。这是在长期运行的对话的有用扩展 会话/持久化上下文。
- » save() 并保证返回一个标识符。如果一个插入必须执行得到标识符(如。“身份”发生器,不是“序列”),该插入发生立即,无论如果你的内部或外部事务。这是一个问题,在长期运行的对话和扩展 会话/持久化上下文。

或者,你可以指定标识符使用重载 版本的 save()。

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex("F");
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

如果你的对象持久化对象(如有关联 小猫 收集在前一个例子),这些对象可以在任何顺序进行持久化你喜欢,除非你有一个非空 约束在一个外键列。有 从来没有一个违反外键约束的风险。然而,你可能会违反 非空 约束如果你 save() 以错误的顺序的对象。

通常你不打扰这个细节,你通常会使用 Hibernate的 过渡持久性 特性来拯救 关联的对象自动。然后,即使 非空 约束违反不发生 - Hibernate将采取 所有的东西都整理好。传递坚持是在稍后讨论这一章。

11.3. 加载一个对象

这个 load() 方法 会话 提供一种方式来获取持久实例如果你知道它 标识符。load() 需要一个类对象和加载 国家进入一个新实例化该类的实例在一个持久 状态。

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );
```

或者,您可以加载状态到一个给定的实例:

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

要知道 load() 将抛出一个不可恢复的 例外,如果没有匹配的数据库行。如果类映射 与一个代理, load() 只返回一个未初始化的 代理,不能真正触及数据库直到你调用一个方法的 代理。这是有用的,如果你想创建一个关联到一个 对象没有实际加载它从数据库。它还允许 加载多个实例作为一个批处理如果 批量大小 是类定义的映射。

如果你不确定一个匹配行存在,你应该使用 这个 get() 方法将数据库立即 并返回null如果没有匹配的行。

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

你甚至可以加载一个对象使用一个SQL 选择..... 对于 更新,使用一个 LockMode。看到API 文档了解更多信息。

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

任何相关的实例或包含集合将 不被选择 更新,除非 你决定指定 锁 或 所有 作为一个级联风格协会。

它是可能的重新加载一个对象及其集合在任何 时间,使用 refresh() 法。这是有用,当 数据库触发器用于初始化的一些属性 对象。

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

多少Hibernate负载从数据库和SQL多少 选择 它将使用s? 这取决于 抓取策略 。 这是解释 20.1节, “抓取策略” 。

11.4. 查询

如果你不知道这个标识符的对象你正在寻找 因为,你需要一个查询。 Hibernate支持一个易于使用的但强大的 面向对象的查询语言(HQL)。 对于编程查询创建, Hibernate支持复杂的标准和示例查询功能(QBC 和QBE)。 你也可以表达你的原生SQL查询你的 数据库,可选支持从Hibernate为结果集转换 成对象。

11 4 1. 执行查询

HQL和原生SQL查询是代表的一个实例 org.hibernate查询 。 这个接口提供了方法 对于参数绑定,结果集处理和执行的 实际查询。 你总是得到一个 查询 使用 电流 会话 :

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

执行一个查询通常是通过调用 列表() 。 查询的结果将被加载到一个集合中完全 内存。 实体实例检索到一个查询是在持续状态。 这个 uniqueResult() 方法提供了一个快捷方式如果你 知道你的查询只会返回一个单一的对象。 查询,利用 即时抓取的集合通常返回副本的根 对象,但随着他们的集合初始化。 你可以过滤这些 副本通过 集 。

11 4 1 1. 迭代结果

有时候,你可以获得更好的性能 执行这个查询使用 迭代() 法。 这通常会是这样的如果你认为实际的实体 查询将返回的实例已经在会话或 二级缓存。 如果他们不是已经缓存, 迭代() 将慢于 列表() 和可能需要许多数据库安打为一个 简单的查询,通常 1 对于最初的选择 这只返回标识符,然后呢 n 额外的 选择初始化实例。

```
// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeliness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next(); // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

11 4 1 2. 查询,该查询返回的元组

Hibernate查询返回的元组对象有时。 每个元组 返回一个数组:

```
Iterator kittensAndMothers = sess.createQuery(
    "select kitten, mother from Cat kitten join kitten.mother mother")
    .list()
    .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten = (Cat) tuple[0];
    Cat mother = (Cat) tuple[1];
    ....
}
```

11 4 1 3. 标量结果

查询可以指定一个属性的一个类 选择 条款。他们甚至可以调用SQL骨料 功能。属性或骨料被视为“标量”的结果 而不是实体在持久状态。

```
Iterator results = sess.createQuery(
    "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
    "group by cat.color")
    .list()
    .iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

11 4 1 4. 绑定参数

方法 查询 提供绑定 值来命名参数或jdbc风格 吗? 参数。与JDBC,Hibernate数字参数 从零。标识符的命名参数的形式 :名字 在查询字符串中。命名的优点 参数如下:

- » 命名参数是迟钝的顺序发生 查询字符串
- » 他们可以进行多次在同一查询
- » 他们具备

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

11 4 1 5. 分页

如果你需要指定界限在你的结果集,即 最大行数你想检索和/或第一行你 要检索,可以使用的方法 查询 接口:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Hibernate知道如何翻译这个限制查询到 原生SQL的数据库管理系统。

11 4 1 6. 可滚动的迭代

如果您的JDBC驱动程序支持可滚动 ResultSet 年代, 查询 接口 可以用来获得一个吗 ScrollableResults 对象 这允许灵活的导航的查询结果。

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
    "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list of cats by name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
}
```

```

while ( cats.scroll(PAGE_SIZE) );

// Now get the first page of cats
pageOfCats = new ArrayList();
cats.beforeFirst();
int i=0;
while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );
}
cats.close()

```

注意,一个开放数据库连接和光标是需要 这个功能。 使用 `setMaxResult()` / `setFirstResult()` 如果你需要离线分页功能。

11 4 1 7. 外化命名查询

查询也可以配置为所谓的命名查询使用 注释或Hibernate映射文件。 `@NamedQuery` 和 `@NamedQueries` 可以定义在类级吗 例11.1, “定义一个命名查询使用 `@NamedQuery` ”。 然而他们的 定义的全局会话工厂/实体管理器工厂 范围。 定义一个命名查询是通过它的名称和实际的查询 字符串。

例11.1. 定义一个命名查询使用 `@NamedQuery`

```

@Entity
@NamedQuery(name="night.moreRecentThan", query="select n from Night n where n.date >= :date")
public class Night {
    ...
}

public class MyDao {
    doStuff() {
        Query q = s.getNamedQuery("night.moreRecentThan");
        q.setDate( "date", aMonthAgo );
        List results = q.list();
        ...
    }
    ...
}

```

使用映射文件可以配置使用 `<查询>` 节点。 记住要使用一个 `CDATA` 部分如果查询包含字符 这可以解释为标记。

例11.2. 定义一个命名查询使用 `<查询>`

```

<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
    where cat.name = ?
    and cat.weight > ?
]]></query>

```

参数绑定和执行完成编程方式见 在 例11.3, “参数绑定的命名查询” 。

例11.3. 参数绑定的命名查询

```

Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();

```

实际的程序代码是独立的查询语言 这是使用。 您还可以定义本地SQL查询元数据,或 迁移现有的查询,把他们在Hibernate映射 文件。

还请注意,一个查询在一个声明 `<hibernate映射>` 元素需要一个全球 独特的名称查询,查询时声明在一个 `<类>` 元素是由独特的自动 我们可以通过按下的类的完全限定名。 例如 `eg.Cat.ByNameAndMaximumWeight` 。

11 4 2. 过滤收集

一个集合 过滤器 是一种特殊类型的吗 查询,可以应用于一个持久的集合或数组。 查询 字符串可以参考 这 ,这意味着当前的集合的元素。

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
);
```

返回的集合被认为是一个包,是一个拷贝的 给定集合。 原来的集合是不能修改的。 这是 与含意的名字 “过滤器” ,但符合 预期行为。

观察过滤不需要的 从 条款,尽管他们可以有一个如果需要。 过滤器并不局限 返回集合元素本身。

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue")
    .list();
```

甚至一个空过滤器查询非常有用,例如加载的一个子集 元素在一个大的集合:

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
    .setFirstResult(0).setMaxResults(10)
    .list();
```

11 4 3. 标准查询

HQL非常强大,但是一些开发人员更喜欢建造 查询动态使用面向对象的API,而不是建筑 查询字符串。 Hibernate提供了一个直观的 标准 查询API对于这些情况下:

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Restrictions.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

这个 标准 和相关的 例子 API将更详细地讨论在 [第十七章, 标准查询](#) 。

11 4 4. 原生SQL查询

你可以表达一个查询在SQL中使用 createSQLQuery() 和让Hibernate管理映射 从结果集对象。 你可以在任何时间打电话 会话连接() 和使用JDBC 连接 直接。 如果您选择使用 Hibernate API,您必须附上SQL别名在括号内:

```
List cats = session.createSQLQuery("SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10")
    .addEntity("cat", Cat.class)
    .list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
    "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10")
    .addEntity("cat", Cat.class)
    .list();
```

SQL查询可以包含命名和位置参数,就像 Hibernate查询。 更多的信息关于原生SQL查询 Hibernate中可以找到 [第18章, 原生SQL](#) 。

11.5. 修改持久化对象

事务性持久实例 (即。 对象加载、保存,创建或查询的 会话)可以被应用程序,和任何 变化将持续持久状态的时候 会话 是 脸红。 这是 在本章后面讨论。 不需要调用一个特定的 方法(如 Update() ,这有一个不同的目的) 让你的修改持久。 最直接的方法更新对象的状态是 load() 它然后 直接对其进行操作而 会话 是 开放:

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush(); // changes to cat are automatically detected and persisted
```

有时这种编程模型是低效的,因为它需要在 相同的会话都一个SQL 选择 加载一个对象 和一个SQL 更新 以保存其更新的状态。 Hibernate提供了另一种方法通过使用分离实例。

11.6. 修改分离对象

许多应用程序需要在一个事务中检索对象, 发送到UI层进行操作,然后将更改保存在一个新的 事务。 应用程序使用这种方法在一个 性环境通常使用版本化的数据,以保证 隔离的 “长” 的工作单元。

Hibernate支持这个模型提供的回贴 分离实例使用 会话更新() 或 会话合并() 方法:

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat); // update cat
secondSession.update(mate); // update mate
```

如果 猫 与标识符 catId 已经被加载 secondSession 当应用程序试图重新接上 它,一个异常被抛出。

使用 Update() 如果你确信会话 不包含一个已经持久化实例相同的标识符。 使用 merge() 如果你想合并你的修改在 任何时候不考虑状态的会话。 在其他 话说, Update() 通常是第一个方法你会吗 打电话叫一个新的会话,确保回贴的分离 实例是第一操作执行的。

应用程序应该单独 Update() 分离实例,从给定的可分离的实例 只有 如果它希望他们的状态更新。 这可以 会自动使用 过渡持久性。 看到 11.11节,“过渡持久性” 为更多的信息。

这个 锁() 方法还允许应用程序 再结合一个对象与一个新的会话。 然而,分离实例 必须修改的。

```
//just reassociate:
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
sess.lock(pk, LockMode.UPGRADE);
```

注意, 锁() 可以用于各种 LockMode 年代。 查看API文档和章节 事务处理更多信息。 回贴不是唯一的 usecase为 锁()。

其他模型讨论了很久的工作单元中 13.3节,“乐观并发控制”。

11.7. 自动状态检测

Hibernate用户请求的一个通用方法,要么 节省一个瞬态实例通过生成一个新的标识符或 更新/ reattaches分离实例关联当前标识符。 这个 saveOrUpdate() 方法实现这个 功能。

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat); // update existing state (cat has a non-null id)
secondSession.saveOrUpdate(mate); // save the new instance (mate has a null id)
```

用法和语义的 saveOrUpdate() 似乎 是令人困惑的新用户。 首先,只要你不是试图 用实例从一个会议在另一个新会话,你应该 不需要 使用 Update(), saveOrUpdate(), 或 merge()。 整个应用程序将永远不会使用一些要么 这些方法。

通常 Update() 或 saveOrUpdate() 用于以下吗 场景:

- 应用程序加载一个对象在第一个会话中
- 对象被传递到UI层
- 一些修改的对象
- 对象被传递到业务逻辑层
- 这些修改的应用程序持续通过调用 Update() 在第二个会话中

saveOrUpdate() 执行以下操作:

- 如果对象已经持续在这个会话,做 没有什么
- 如果另一个与对象关联的会话有相同的 标识符,抛出一个异常
- 如果对象没有标识符属性, save() 它
- 如果对象的标识符的值分配给一个新 实例化的对象, save() 它
- 如果对象是由一个版本 <版本> 或 < timestamp >, 版本属性值 是相同的值分配给一个新实例化的对象, save() 它
- 否则 Update() 对象

和 merge() 是不同的:

- 如果有一个持久实例相同的标识符 目前与会话关联起来,复制状态给定的 对象持久性实例上
- 如果没有持久实例目前相关 会话,试图从数据库加载它,或创建一个新的持久 实例
- 持久实例返回
- 给定的实例并不成为与会话关联起来, 它仍然是分离

11.8. 删除持久对象

会话删除() 将删除一个对象的状态吗 从数据库。 你的应用程序,然而,仍然可以保持一个参考 一个被删除的对象。 最好是可以理解的 删除() 作为 做一个持久实例,瞬态。

```
sess.delete(cat);
```

你可以在任何顺序删除对象;没有风险的外键 约束违反。 它仍然有可能违反了 不空 约束在一个外键列,删除对象 错误的订单,例如如果你删除父,但忘了删除 的孩子。

11.9. 在两个不同的数据存储复制对象

它有时是有用的,可以把一个图的持久性 实例和使他们持续在一个不同的数据存储,没有 再生的标识值。

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

这个 ReplicationMode 决定 复制() 将处理冲突与现有的行吗 在数据库:

- ▶ ReplicationMode.IGNORE :忽略了对象 当有一个现有的数据库行相同的标识符
- ▶ ReplicationMode.OVERWRITE :覆盖任何 现有的数据库行相同的标识符
- ▶ ReplicationMode.EXCEPTION :抛出一个 例外如果一个现有的数据库行相同的 标识符
- ▶ ReplicationMode.LATEST_VERSION :覆盖 行如果其版本号比版本号的早些时候 对象,或忽略该对象否则

用例这个特性包括协调数据进入 不同的数据库实例,升级系统配置信息 在产品升级,回滚更改在非酸 交易和更多。

11.10. 冲洗会话

有时 会话 将执行SQL 语句需要同步的JDBC连接的状态的 对象的状态保存在内存中。 这个过程称为 冲洗,发生在默认情况下 在下面 点:

- ▶ 一些查询执行之前
- ▶ 从 org.hibernate.事务提交()
- ▶ 从 会话冲洗()

发出的SQL语句按下列顺序:

1. 所有实体插入在同一个订单对应的 对象保存使用 会话保存()
2. 所有实体更新
3. 所有收集删除
4. 所有集合元素删除、更新和插入
5. 所有收集插入
6. 删除所有实体在同一订单对应的对象 被删除使用 会话删除()

一个例外是,对象使用 原生 id 代时插入保存。

除非你明确 flush(),有 绝对不能保证 当 这个 会话 执行JDBC调用,只有 秩序 他们正在执行。 然而,冬眠 并保证 查询列表(.) 永远不会 返回过期或不正确的数据。

这是有可能改变默认的行为,以便冲洗发生 更少的。 这个 FlushMode 类定义了三个 不同的模式:只有平在提交时当 Hibernate 事务 API使用,冲洗自动使用 解释程序,或从不冲洗除非 flush() 是 称为明确。 过去的模式有助于长时间运行的工作单元,一个 会话 保持开放和未连接一个吗 长时间(见 节13 3 2,“延长会话和自动版本”)。

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // allow queries to return stale state

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// might return stale data
```

```

sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();

```

在冲洗,异常可能发生(例如,如果一个DML操作 违反了约束)。 因为涉及到一些处理异常 Hibernate的事务行为的理解,我们将讨论它 [第十三章, 事务和并发性](#)。

11.11. 过渡持久性

它是相当繁琐的保存、删除或重新接上个人 对象,特别是如果你处理一个图相关的对象。 一个 常见的情况是一个父/子关系。 考虑以下 示例:

如果孩子在一个父/子关系将值类型 (如一个集合的地址或字符串),他们的生命周期将取决于 在父母和没有进一步行动需要方便 “级联” 的状态变化。 当父保存,值类型 子对象保存,当父被删除时,孩子会 被删除,等等。 这种方法适用于操作,如移除一个孩子 从集合中。 从值类型对象不能共享 引用,Hibernate将会检测到这些并删除这个孩子的 数据库。

现在考虑同样的场景与父进程和子对象 实体,而不是值类型(例如类别和项目,或父母和孩子 猫)。 的实体都有自己的生命周期 和支持共享引用。 从集合中删除一个实体并不意味着它可以被删除,和有默认情况下没有级联的状态从一个实体到任何其他 相关的实体。 Hibernate没有实现 持久性的 可达性 默认情况下。

对于每个基本操作的Hibernate会话——包括 persist(),saveOrUpdate合并(),删除(),锁(),刷新(), 驱逐(),复制() ——有一个 相应的级联风格。 分别,级联样式命名 创建、合并、保存更新、删除、锁定、刷新、驱逐、复制。 如果你 想要一个操作级联 以及一个协会,你必须注明 在映射文件。 例如:

```
<one-to-one name="person" cascade="persist"/>
```

级联样式我结合在一起:

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

你甚至可以使用 级联= “所有” 指定 所有 级联操作应该沿着 协会。 默认 级联= “什么都没有” 指定 任何操作都是级联。

如果您使用的是annotatons你可能已经注意到 级联 以一个数组的属性 cascadetype 作为一个值。 在JPA的级联的概念 非常类似于过渡持久性和级联的 操作如上所述,但是稍微不同的语义和 级联类型:

- ▶ CascadeType.PERSIST :小瀑布的坚持 (创建)操作相关实体persist()被称为或如果 实体管理
- ▶ CascadeType.MERGE :小瀑布合并 操作相关的实体如果merge()被称为或者实体 管理
- ▶ CascadeType.REMOVE :级联删除 操作相关的实体如果删除()被调用
- ▶ CascadeType.REFRESH: 瀑布刷新 操作相关的实体如果refresh()被称为
- ▶ CascadeType.DETACH: 级联分离的 操作相关的实体如果分离()被调用
- ▶ CascadeType.ALL :上述所有

注意

CascadeType。 所有还包括Hibernate具体操作像 保存更新锁等等.....

一个特殊的级联风格,删除孤儿,适用于 只有一对多关联,指出 删除() 操作应该被应用到任何孩子 对象,它将从该协会。 使用注释没有 CascadeType.DELETE-ORPHAN 等效。 相反,你可以 使用属性 orphanRemoval见 [例11.4, “ onetomany 与 orphanRemoval ”](#)。 如果一个实体是 移除一个 onetomany 集合或一个 反向关联实体从一个 @OneToOne 协会,这个关联实体可以被标记为删除如果 orphanRemoval 被设置为TRUE。

例11.4. onetomany 与 orphanRemoval

```

@Entity
public class Customer {
    private Set<Order> orders;

    @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)
    public Set<Order> getOrders() { return orders; }

    public void setOrders(Set<Order> orders) { this.orders = orders; }

    [...]
}

@Entity
public class Order { ... }

Customer customer = em.find(Customer.class, 1);
Order order = em.find(Order.class, 1);
customer.getOrders().remove(order); //order will be deleted by cascade

```


建议:

- ▶ 它没有通常意义上启用级联 多对一和多对多关联。事实上, @ManyToOne 和 @ManyToMany 不要 甚至提供一个 orphanRemoval 属性。层叠是 常用于一对一和一对多关联。
- ▶ 如果子对象的生命是有限的寿命的 父对象,使它成为一个 生命周期对象 通过 指定 级联= "删除孤儿" (CascadeType.ALL onetomany(级联=, orphanRemoval = true))。
- ▶ 否则,你可能不需要级联在所有。但如果你认为 那你会经常被处理父母和孩子在一起 在同一事务,和你想节省一些打字,考虑使用 级联= "持续、合并、保存更新" 。

映射一个协会(要么是单值的协会,或 收集) 级联= "所有" 标志着协会 一个 父/子 风格的关系, 保存/更新/删除父结果在保存/更新/删除的 孩子或孩子。

此外,仅引用一个孩子从一个持久的父母 会导致孩子的保存/更新。这个比喻是不完整的, 然而。 一个孩子,成为未引用由其家长是 不 自动删除,有一种情况除外 一对多关联映射与 级联= "删除孤儿" 。精确的语义 级联操作一个父/子关系一样 如下:

- ▶ 如果父母传递给 persist() ,所有 孩子们被传递给 persist()
- ▶ 如果父母传递给 merge() ,所有 孩子们被传递给 merge()
- ▶ 如果父母传递给 save() , Update() 或 saveOrUpdate() ,所有 孩子们被传递给 saveOrUpdate()
- ▶ 如果一个瞬态或分离的孩子变成一个引用 持久的家长,它传递给 saveOrUpdate()
- ▶ 如果父母是删除,所有的孩子都传递给 删除()
- ▶ 如果一个孩子是一个持久的引用的家长, 没什么特别的发生 ——应用程序应该 孩子如果有必要显式地删除,除非 级联= "删除孤儿" ,在这种情况下 "孤儿" 的孩子被删除。

最后,请注意,层叠的操作可以应用到一个 对象图在 呼叫时间 或在 冲洗 时间 。所有的操作,如果启用,级联相关 实体可操作执行。 然而, 保存更新 和 删除孤儿 是 对所有相关的实体可传递在冲洗的 会话 。

11.12. 使用元数据

Hibernate需要丰富的元级别模型的实体和价值 类型。 这个模型可以是很有用的应用程序本身。 例如, 应用程序可能使用 Hibernate的元数据来实现 "智能" 深复制算法,了解哪些对象应该被复制(例如。 可变值类型)和哪些对象,不应该(如不可变的值类型和可能相关的实体)。

Hibernate暴露元数据通过 ClassMetadata 和 CollectionMetadata 接口和 类型 层次结构。 元数据接口的实例 可以得到吗 SessionFactory 。

```
Cat fritz = .....;
ClassMetadata catMeta = sessionFactory.getClassMetadata(Cat.class);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// get a Map of all properties which are not collections or associations
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

第十二章。 只读实体

表的内容

12.1. 让持久化实体只读

- 12个1 1. 实体不可变类
- 12个1 2. 加载持久化实体为只读
- 12个1 3. 只读实体加载从HQL查询/标准
- 12个1 4. 做一个持久化实体只读

12.2. 只读属性类型影响

- 12 2 1. 简单属性
- 12 2 2. 单向关联
- 12 2 3. 双向关联

重要

Hibernate的治疗 只读 实体可能 不同于你所可能遇到的其他地方。 不正确的使用 可能会导致意想不到的结果。

当一个实体是只读的:

- Hibernate不脏检查实体的简单 属性或单端关联;
- Hibernate将不会更新简单属性或可更新的 单端关联;
- Hibernate将不会更新版本的只读 如果只是简单的属性或实体单端 可更新的协会正在改变;

在某些方面,Hibernate将只读实体实体一样 不是只读的:

- Hibernate瀑布行动协会 定义在实体映射。
- Hibernate更新版本如果实体有一个 收集与变化,污垢的实体;
- 一个只读实体可以被删除。

即使一个实体不是只读的,其收藏协会可以 会受到影响,如果它包含了一个只读实体。

了解详细的只读实体在不同的影响 属性和关联类型,见 12.2节,“只读属性类型影响”。

对于如何使实体只读,请参阅 12.1节,“让持久化实体只读”

Hibernate进行一些优化为只读实体:

- 它节省了执行时间不脏检查简单属性或 单端关联。
- 它节省了内存数据库通过删除快照。

12.1. 让持久化实体只读

只有持久化实体可以只读。 瞬态和 分离的实体必须放在之前持久状态 可以只读。

Hibernate提供了以下方法让持久化实体只读:

- 你可以映射一个实体类 不变的 ; 当一个实体的一个不可变类持久不变, Hibernate自动使其只读的。 看到 节12 1 1” ,[实体的不可变类](#)” 详情
- 您可以更改默认这样实体加载 在会议上通过Hibernate自动 使只读;看到 节12 1 2、[“加载持久化实体为只读”](#) 详情
- 你可以做一个HQL查询或标准只读所以 ,实体加载当查询或标准执行, 卷轴,或者迭代,自动 使只读;看到 节12 1 3、[“只读实体加载从HQL查询/标准”](#) 详情
- 你可以做一个持久化实体已经在 在会话中只读;请参阅 节12 1 4” ,[使得一个持久化实体只读](#)” 详情

12个1 1. 实体不可变类

当一个实体的实例是一个不可变类 持久,Hibernate自动使其只读的。

一个实体的一个不可变类可以创建 和删除相同的作为一个实体的一个可变类。

Hibernate将持久化实体的一个不可变的 类一样只读持久化实体 一个可变类。 唯一的例外是, Hibernate不会允许一个实体的一个不可变的 类是改变了所以它不是只读的。

12个1 2. 加载持久化实体为只读

注意

实体的不可变类会自动加载 为只读。

改变默认的行为所以Hibernate加载实体 可变类的实例到会话和自动 使他们只读,电话:

```
Session.setDefaultReadOnly( true );
```

改变默认的回所以实体加载Hibernate不 使只读,电话:

```
Session.setDefaultReadOnly( false );
```

你可以确定当前的设置通过调用:

```
Session.isDefaultReadOnly();
```

如果Session.isDefaultReadOnly()返回true,实体加载 以下将自动只读:

- 会话负荷()
- 会话获得()
- 会话合并()
- 执行、滚动、或迭代HQL查询和 标准,以覆盖该设置为一个特定的 HQL查询或标准见 节12 1 3、[“只读实体加载从HQL查询/标准”](#)

改变这种默认不影响:

- » 持久化实体已经在会话默认了
- » 持久化实体,通过刷新 会话刷新(),一个刷新持久 实体只能只读如果是 只读前刷新
- » 持久化实体添加应用程序通过 会话persist(),会话保存()和会话更新() Session.saveOrUpdate()

12个1 3。 只读实体加载从HQL查询/标准

注意

实体的不可变类会自动加载 为只读。

如果Session.isDefaultReadOnly()返回false(默认) 当一个HQL查询或标准执行,然后实体 和代理的可变类加载查询 不是只读的。

你可以覆盖这一行为,以便加载实体和代理 通过一个HQL查询或标准将自动只读。

对于一个HQL查询,电话:

```
Query.setReadOnly( true );
```

查询。 setReadOnly(真正的) 之前必须调用 查询列表(), Query.uniqueResult(), 查询滚动(),或 查询迭代()

对于一个HQL标准,电话:

```
Criteria.setReadOnly( true );
```

标准。 setReadOnly(真正的) 之前必须调用 标准列表(), Criteria.uniqueResult(), 或 标准滚动()

实体和代理中存在的会话之前返回 通过一个HQL查询或标准不受影响。

未初始化的持久化集合查询返回 不受影响的。 后来,当集合初始化, 实体加载到会话将只读如果 Session.isDefaultReadOnly()返回true。

使用 查询。 setReadOnly(真正的) 或 标准。 setReadOnly(真正的) 工作良好 当一个HQL查询或标准加载所有的实体和 initializes所有的代理和应用程序集合 需要只读。

如果不可能加载和初始化所有 必要的实体在一个单独的查询或标准, 你可以暂时改变会话默认加载 实体作为只读查询执行之前。 然后您可以显式初始化代理和集合 在恢复会话违约。

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

setDefaultReadOnly( true );
Contract contract =
    ( Contract ) session.createQuery(
        "from Contract where customerName = 'Sherman'" )
        .uniqueResult();
Hibernate.initialize( contract.getPlan() );
Hibernate.initialize( contract.getVariations() );
Hibernate.initialize( contract.getNotes() );
setDefaultReadOnly( false );
...
tx.commit();
session.close();
```

如果Session.isDefaultReadOnly()返回true,那么你就可以 使用查询。 setReadOnly(假)和标准。 setReadOnly(假) 覆盖这个会话设置和加载实体 不是只读的。

12个1 4。 做一个持久化实体只读

注意

持久化实体的不可变类自动 使只读。

做一个持久化实体或代理只读,电话:

```
Session.setReadOnly(entityOrProxy, true)
```

改变一个只读实体或代理的一个可变类 它不再是只读的,电话:

```
Session.setReadOnly(entityOrProxy, false)
```

重要

当一个只读实体或代理是改变了所以它不再是 只读,Hibernate假设当前的状态 只读实体是符合其数据库表示。 如果这不是真的,那么任何非冲洗之前更改 或者在实体是只读的,将被忽略。

扔掉不脸红的变化和使持久化实体 符合它的数据库表示,电话:

```
session.refresh( entity );
```

更改刷新之前或在实体 是只读的,使数据库表示吗 符合当前状态的持久性 实体:

```
// evict the read-only entity so it is detached
session.evict( entity );

// make the detached entity (with the non-flushed changes) persistent
session.update( entity );

// now entity is no longer read-only and its changes can be flushed
s.flush();
```

12.2. 只读属性类型影响

下面的表总结了不同的属性类型 受制作一个实体只读。

表12.1. 影响房地产的只读实体类型

财产/协会类型	更改刷新到数据库吗?
简单 (节12 2 1, "简单属性")	没有*
单向一对一	没有*
单向多对一 (节12 2 2 1, "单向一对一和多对一的")	没有*
单向一对多	是的
单向多对多 (节12 2 2 2, "单向一对多和多对多")	是的
双向一对一 (节12 2 3 1, "双向一对一")	只有拥有实体不是只读*
双向一对多或多对一的	
逆收集	只有添加/删除实体不是只读*
非逆收集 (节12 2 3 2, "双向一对多或多对一的")	是的
双向多对多 (节12 2 3 3, "双向多对多")	是的

*行为是不同的实体拥有财产时/协会 是只读的,而当它不是只读的。

12 2 1. 简单属性

当一个持久化对象是只读的,Hibernate不 脏检查简单属性。

Hibernate不会同步简单属性状态的改变 到数据库。 如果你有自动版本,Hibernate 不会增加版本如果任何简单的属性变化。

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

// get a contract and make it read-only
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );

// contract.getCustomerName() is "Sherman"
contract.setCustomerName( "Yogi" );
```

```

tx.commit();

tx = session.beginTransaction();

contract = ( Contract ) session.get( Contract.class, contractId );
// contract.getCustomerName() is still "Sherman"
...
tx.commit();
session.close();

```

12 2 2. 单向关联

12 2 2 1. 单向一对一和多对一的

Hibernate将单向一对一和多对一 协会以同样的方式当拥有实体 只读的。

我们使用术语 单向单端 协会 当指功能 这是共同的单向一对一和多对一 协会。

Hibernate不脏检查单向单端 协会当拥有实体是只读的。

如果你改变一个只读实体的引用 单向单端协会为空, 或指一个不同的实体,这种变化 将不会刷新到数据库。

注意

如果一个实体是一个不可变类, 那么它的引用单向单端 联想必须分配当 第一次被创建的实体。 因为实体 自动成为只读的,这些引用可以 没有被更新。

如果自动版本使用,Hibernate不会 增加版本由于本地修改 单向单端关联。

在下面的例子中,合同有一个单向 多对一关联与计划。 合同瀑布保存并 更新操作的协会。

以下内容显示,改变一个只读实体的 多对一关联参考空没有效果 在实体的数据库表示。

```

// get a contract with an existing plan;
// make the contract read-only and set its plan to null
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
contract.setPlan( null );
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );

// contract.getPlan() still refers to the original plan;

tx.commit();
session.close();

```

以下说明,尽管 一个更新一个只读实体的多对一 协会没有影响实体的 数据库表示,冲洗还小瀑布 这个保存更新操作到本地 改变了协会。

```

// get a contract with an existing plan;
// make the contract read-only and change to a new plan
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
Plan newPlan = new Plan( "new plan" );
contract.setPlan( newPlan );
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );
newPlan = ( Contract ) session.get( Plan.class, newPlan.getId() );

// contract.getPlan() still refers to the original plan;
// newPlan is non-null because it was persisted when
// the previous transaction was committed;

tx.commit();
session.close();

```

12 2 2 2. 单向一对多和多对多

Hibernate将单向一对多 和多对多关联属于一个只读的 实体时一样属于一个实体,不是 只读的。

Hibernate脏检查单向一对多和 多对多关联;

集合可以包含实体 是只读的,以及实体吗 这不是只读的。

实体可以加入或移出 收集;修改刷新到数据库。

如果自动版本使用,Hibernate将 更新版本变动由于集合 如果他们脏的拥有实体。

12 2 3。双向关联

12 2 3 1。双向一对一

如果一个只读实体拥有一个双向 一对一关联:

- » Hibernate不脏检查协会。
- » 更新,改变协会参考 为null或引用不同的实体 将不会刷新到数据库。
- » 如果自动版本使用,Hibernate不会 增加版本由于本地修改 该协会。

注意

如果一个实体是一个不可变类,而且它拥有一个双向一对一 协会,那么它的引用必须 指定当实体第一次被创建。 因为实体是自动生成 只读的,这些引用不能被更新。

当老板不是只读的,Hibernate对待 一个协会和一个只读实体相同 当这个协会是一个实体 不是只读的。

12 2 3 2。双向一对多或多对一的

一个只读实体没有影响双向 一对多/多对一关联如果:

- » 只读实体在一对多的一面 使用一个逆收集;
- » 只读实体在一对多的一面 使用非逆收集;
- » 一对多侧使用非逆集合 包含只读实体

当一对多侧使用一个逆收集:

- » 一个只读实体只能被添加到集合 当它被创建;
- » 一个只读实体只能删除的 收集一个孤儿删除或通过显式 删除实体。

12 2 3 3。双向多对多

Hibernate将双向多对多 属于一个只读实体关联的 时一样属于一个实体,不是 只读的。

Hibernate脏检查双向多对多 协会。

收藏协会的两边 可以包含实体是只读的, 作为实体,不是只读的。

实体是双方的添加和删除 集合的;修改刷新到 数据库。

如果自动版本使用,Hibernate将 更新版本由于双方的变化 收集如果他们脏实体拥有 各自的集合。

第13章。事务和并发性

表的内容

13.1。会话和事务范围

- 13 1 1。工作单元
- 13 1 2。长对话
- 13.1.3节。考虑对象的身份
- 13 1 4。常见问题

13.2。数据库事务界定

- 13 2 1。不受管理的环境下
- 13 2 2。使用JTA
- 13 2 3。异常处理

13.2.4. 事务超时

13.3. 乐观并发控制

13.3.1. 应用版本检查

13.3.2. 扩展的会话和自动版本

13.3.3. 分离对象和自动版本

13.3.4. 定制自动版本

13.4. 悲观锁

13.5. 连接释放模式

最重要的一点是,Hibernate和并发控制它 容易理解。 Hibernate直接使用JDBC连接和JTA资源没有 添加任何额外的锁定行为。 建议您花一些时间 JDBC,ANSI,事务隔离规范你的数据库管理系统。

Hibernate不锁对象在内存中。 您的应用程序可以预期行为 定义你的数据库事务隔离级别。 通过 会话 ,这也是一个以缓存,Hibernate 提供了可重复读取由标识符查找和实体的查询和不是 报告查询,返回标量值。

除了版本自动乐观并发控制,Hibernate也 提供,使用 选择更新 语法,一个(小)API为悲观锁定的行。 乐观并发控制和 这个API是在本章后面讨论。

并发控制的讨论在Hibernate始于的粒度 配置,SessionFactory ,和 会话 以及数据库事务和长对话。

13.1. 会话和事务范围

一个 SessionFactory 是一种创建开销很大,线程安全对象,打算由所有应用程序线程共享。 它是创建一次,通常在 应用程序启动时,从一个 配置 实例。

一个 会话 是廉价、非线性安全的对象,应该吗 使用一次,然后丢弃:单个请求,对话或单个的工作单元。 一个 会话 不会获得JDBC吗 连接,或 数据源,除非它是必要的。 它不会消耗任何 直到使用资源。

为了减少锁争用的数据库,一个数据库事务必须尽可能地短。 长数据库事务将防止应用程序从定标 到一个高度并发负载。 我不建议你持有 数据库事务期间开放用户思考时间,直到工作单元 完整的。

的范围是什么单位工作? 可以一个Hibernate 会话 跨几个数据库事务,或这是一个一对一的关系的范围? 当 你应该打开和关闭吗 会话 和你如何的标定 数据库事务边界? 解决这些问题是在以下部分。

13.1.1. 工作单元

首先,让我们定义一个工作单元。 一个工作单元是一个 设计模式被马丁作为 “(维护)的一个列表对象受业务 事务和坐标写作的变化和解决并发问题。” [PoEAA] 换句话说,它的一系列操作我们希望开展 对数据库在一起。 基本上,这是一个交易,虽然完成一个工作单元通常会跨越多个 物理数据库事务(参见 节13.1.2,“长对话”)。 所以真的我们正在谈论一个更抽象的概念 事务。 术语“商业交易”有时也 用代替的工作单元。

不使用 会话/操作 反模式: 不要打开和关闭吗 会话 对于每一个简单的数据库调用 一个线程。 这同样适用于数据库事务。 数据库调用 在一个应用程序都是使用一个计划序列,他们被分组到原子 的工作单元。 这也意味着,在每一个自动提交 SQL语句是无用的在一个应用程序,这个模式是用于SQL 控制台工作。 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 沟通与数据库已经发生在一个事务。 阅读数据自动提交行为 应该避免,因为许多小交易不太可能表现得比吗 一个清晰定义的工作单元。 后者也更易于维护 和可扩展的。

最常见的模式在一个多用户的客户机/服务器应用程序 使用“按请求会话”模式。 在这个模型中,一个来自客户机的请求 被发送到服务器,Hibernate持久性层运行。 一个新的Hibernate 会话 是开了,所有数据库操作都执行本单位 的工作。 在完成了工作,而且一旦响应客户机已经准备好了,会话是刷新和关闭。 使用一个数据库事务 客户服务请求,启动和提交它当你打开和关闭会话。 两者的关系是一对一和这个 模型是一个完美的适合许多应用程序。

挑战在于实现。 Hibernate提供了内置的管理“当前会话”来简化这个模式。 开始 当一个服务器请求的事务要处理和结束事务 在响应被发送到客户端。 常见的解决方案是 ServletFilter ,AOP拦截器与一个 切入点在服务方法,或一个代理/拦截容器。 EJB容器 是一个标准化的方法来实现横切方面如事务 界定EJB会话bean,声明与CMT。 如果你 使用程序性事务界定,为便于使用和代码可移植性使用Hibernate 事务 API在本章后面所示。

您的应用程序代码可以访问一个“当前会话”来处理请求 通过调用 SessionFactory.getCurrentSession() 。 你总是会得到一个 会话 范围 到当前数据库事务。 这必须配置为要么 本地资源或JTA环境,看到 2.3节,“上下文会话”。

您可以扩展的范围 会话 和 数据库事务,直到“查看已呈现”。 这是特别有用 在servlet应用程序利用一个单独的渲染阶段之后的请求 被处理。 扩展数据库事务直到视图渲染,是通过实施 你自己的拦截器。 然而,这将是困难的 如果你依赖ejb和容器管理的事务。 一个 交易将完成当EJB方法返回,在呈现之前的任何 视图可以开始。 看到Hibernate网站和论坛相关的提示和示例 这 公开会议针对 模式。

13.1.2. 长对话

使用“按请求会话”模式的模式并不是唯一的方式设计 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的 交叉与数据库访问。 在web和企业应用程序,它是 不可接受的数据库事务跨越一个用户交互。 考虑以下 示例:

- » 的第一个屏幕将会打开一个对话框。 数据被用户看到已经载入 一个特定的 会话 和数据库事务。 用户是免费的 修改对象。

- ▶ 用户点击“保存”后5分钟,预计他们的修改做出持久。用户还希望他们是唯一的人编辑这个信息和没有发生冲突的修改。

从用户的角度,我们把这种工作单元的一个长时间运行的谈话或应用程序事务。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交易开放在用户思考时间,与数据库中的锁来防止并发修改并保证隔离和原子性。这是一种反模式,自锁争用不会允许应用程序规模与并发用户的数量。

你必须使用多个数据库事务来实现对话。在这种情况下,保持隔离的业务流程成为应用程序层的部分责任。一个对话通常跨越几个数据库事务。它将原子如果只有一个这些数据库事务(最后一个)商店更新的数据。所有其他简单地读取数据(例如,在一个向导样式对话框生成几个请求/响应周期)。这是容易实现,但听起来,特别是如果你利用一些Hibernate的特点:

- ▶ 自动版本:Hibernate可以执行自动乐观并发控制你。它可以自动检测如果一个并发修改发生在用户思考时间。检查这个在谈话结束。
- ▶ 分离对象:如果你决定使用使用“按请求会话”模式,所有加载实例在分离的状态将在用户思考时间。Hibernate允许你将对象和保存修改。这个模式称为每个请求的会话与分离对象。自动版本控制是用来隔离并发修改。
- ▶ 扩展(或长期)会话:冬眠会话可以断开与底层JDBC吗连接在数据库事务已经提交,重新连接当一个新的客户端请求发生时。此模式被称为会话/谈话并使即使回贴不必要的。自动版本用于隔离并发修改和会话不会被允许自动刷新,但明确。

两个每个请求的会话与分离对象和会话/谈话有优点和缺点。这些缺点是在本章后面讨论上下文中的乐观并发控制。

13.1.3节。考虑对象的身份

一个应用程序可以同时访问相同的持续状态在两个不同会话年代。然而,持久化类的一个实例从来不是分给两吗会话实例。正是因为这一原因,有两个不同身份的概念:

数据库身份

```
foo.getId().equals(酒吧.getId())
```

JVM身份

```
foo == 酒吧
```

连接到一个对象特定会话(即,在的范围会话),这两个概念是等价的,JVM身份数据库身份保证了Hibernate。当应用程序可能并发访问“相同”(持久化标识)业务对象在两个不同的会议上,两个实例将实际上是“不同”(JVM身份)。冲突用一种乐观的方式解决和自动版本在冲洗/提交时间。

这种方法叶子Hibernate和数据库担心并发。它还提供了最好的可伸缩性,因为保证身份在单线程的工作单元意味着它不需要昂贵的锁或其他的同步方法。应用程序不需要同步任何业务对象,只要保持一个线程每会话。在会话应用程序可以安全地使用 == 比较对象。

然而,一个应用程序,该应用程序使用 == 外的会话可能会产生意想不到的结果。这可能发生,甚至在一些意想不到的地方。例如,如果你把两个分离实例到相同的集,都可能具有相同的数据库的身份(即,他们表示相同的行)。JVM的身份,然而,在定义上是没有任何的保证为实例在一个分离的状态。开发人员必须覆盖 equals() 和 hashCode() 方法在持久化类和实现自己的对象概念平等。有一点要注意:不要使用数据库标识符来实现平等。使用一个业务键结合的独特,通常不可变的,属性。数据库标识符将改变如果一个瞬态对象了持久。如果瞬态实例(通常连同分离实例)被固定在一个集,改变了hashCode打破了合同集。属性为业务键不需要数据库的主键一样稳定;你必须保证稳定,只要对象是相同的吗集。看到Hibernate网站更彻底地讨论这个问题。请注意,这不是一个Hibernate的问题,而只是说明Java对象的身份与平等实现。

13 1 4。常见问题

不要使用反模式会话/用户会话或会话/应用程序(然而,罕见的例外这条规则)。下面的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。事情与此同时,像HTTP请求、会话bean或摆动的工人,将导致种族条件如果会话实例共享。如果你保持你的Hibernate会话在你的HttpSession(这是讨论在之后的章节),你应该考虑同步访问您的Http会话。否则,一个用户,点击重新加载速度不够快可以使用相同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(后面将更详细的讨论在这一章)。如果你会话被绑定到应用程序,你必须停止吗应用程序。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(看和检查脏状态Hibernate)。如果你保持开放的时间长或简单的负载太多的数据,它将增长不断,直到你得到一个OutOfMemoryException。一个解决办法是调用 clear() 和 驱逐() 管理会话缓存,但是你应该考虑存储过程如果你需要大规模的数据操作。一些解决方案中所示第十五章,批处理。保持一个会话开放期间用户会话也意味着更高的概率的失效数据。

13.2。数据库事务界定

数据库或系统,事务边界总是必要的。没有沟通数据库可以发生在数据库事务之外(这似乎混淆了许多开发人员他已经习惯于自动提交模式)。总是使用清晰的事务边界,甚至只读操作。取决于你的隔离级别和数据库功能可能不是这样的,是必需的,但是如果你没有缺点总是限定交易明确。当然,一个数据库事务是要表现得比许多小事务,甚至阅读数据。

Hibernate应用程序可以运行在非受管(即。 、独立、简单的Web -或Swing应用程序) 和管理J2EE环境。 在非受管环境中,Hibernate通常负责 自己的数据库连接池。 应用程序开发人员必须手动设置事务 边界(开始、提交或回滚数据库事务)自己。 一个托管环境 通常提供容器管理的事务(CMT),交易会以声明的方式定义 (在EJB会话bean的部署描述符,例如)。 程序性事务界定是 然后不再必要。

然而,通常需要你保持你的持久层移植到不同非受管 本地资源环境和系统能够依靠JTA但使用BMT代替CMT。 在这两种情况下使用程序性事务界定。 Hibernate提供了一个包装 API称为 事务 这转化为本地事务系统的 您的部署环境。 这个API实际上是可选的,但我们强烈鼓励其使用 除非你是CMT的会话bean。

结束 会话 通常涉及四个明显的阶段:

- 冲洗会话
- 提交事务
- 关闭会话
- 处理异常

我们讨论了冲洗会话之前,所以现在我们将参观一下事务 界定和异常处理在两个管理的或不受管理的环境。

13 2 1. 不受管理的环境下

如果一个Hibernate持久性层运行在非受管环境中,数据库连接 通常由简单(即。 ,非datasource)连接池从哪个 Hibernate获得所需的连接。 会话/事务处理习语看起来 这样的:

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

你不需要 flush() 这个 会话 明确: 调用 commit() 自动触发同步不同 在 11.10节,“冲洗会话” 对于会话。 一个叫 close() 标志着一个会话结束。 主要的含义 的 close() 是JDBC连接将被放弃的吗 会话。 这个Java代码并运行在两个非受管便携式和JTA环境。

概述早些时候,一个更灵活的解决方案是Hibernate内置的“当前会话” 上下文 管理:

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();

    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

你不会看到这些代码片段在一个常规的应用程序; 致命的(系统)例外总是应该抓住“顶级”。 换句话说, 执行的代码调用Hibernate在持久层,代码处理 RuntimeException (而且通常只能清理和退出),都在 不同的层。 当前的上下文管理通过Hibernate可以显著 简化这个设计通过访问 SessionFactory 。 异常处理是在本章后面讨论。

你应该选择 org.hibernate.transaction.JDBCTransactionFactory , 这是默认的,和第二个示例选择吗 “线程” 作为你的hibernate当前会话上下文类。

13 2 2. 使用JTA

如果你的持久层运行在应用服务器(例如,后面EJB会话bean), 每个数据源连接通过Hibernate将自动获得全球的一部分 JTA事务。 你也可以安装一个独立的JTA实现和使用它没有 EJB。 Hibernate提供了两种策略对JTA集成。

如果你使用bean管理的事务(BMT),Hibernate会告诉应用程序服务器开始 和结束BMT事务如果使用 事务 API。 这个 事务管理代码相同,不受管理的环境下。

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
```

```

try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}

```

如果你想使用一个事务绑定 会话,即 `getCurrentSession()` 功能便于上下文传播,使用JTA UserTransaction API直接:

```

// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
        .lookup("java:comp/UserTransaction");

    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}

```

与CMT,事务划分完成后在会话bean的部署描述符,而不是编程。代码是减少到:

```

// CMT idiom
Session sess = factory.getCurrentSession();

// do some work
...

```

在一个CMT / EJB,甚至发生自动回滚。一个未处理的 `RuntimeException` 抛出的一个会话bean方法告诉容器设置全局事务回滚。你不需要使用Hibernate 事务 API在 所有与BMT或CMT,你会自动传播的“当前”会话绑定到 事务。

当配置Hibernate的事务工厂,选择 `org.hibernate.transaction.JTATransactionFactory` 如果你使用JTA直接(BMT),`org.hibernate.transaction.CMTTransactionFactory` CMT的会话bean。记得还设置 `manager_lookup_class`。确保 那你 `hibernate`当前会话上下文类 要么是复原(向后 兼容性),或设置为“jta”。

这个 `getCurrentSession()` 操作有一个缺点在JTA环境。有一点要注意使用 在声明中 连接释放 模式,然后使用它在默认情况下。由于限制了JTA规范,它不是 可能对于Hibernate自动清理任何未关的 `ScrollableResults` 或 迭代器 实例返回的 `滚动()` 或 `迭代()`。你必须 释放底层数据库 光标通过调用 `ScrollableResults.close()` 或 `hibernate`关闭(迭代器) 显式地从一个 最后 块。大多数应用程序可以很容易地避免使用 `滚动()` 或 `迭代()` 从JTA或者CMT代码。))

13 2 3. 异常处理

如果 会话 抛出一个异常,包括任何 `SQLException`,立即回滚数据库 事务,调用 `会话关闭()` 和丢弃的 会话 实例。某些方法的 会话 将 不 离开会话保持一致的状态。没有 异常抛出的Hibernate可视为可恢复。确保 会话 通过调用将被关闭 `close()` 在一个 最后 块。

这个 `HibernateException`,它封装了大部分的错误 可以发生在一个Hibernate持久性层,是一个未经检查的异常。这不是 在旧版本的冬眠。在我们看来,我们不应该迫使应用程序 开发人员抓住一个不可恢复的异常较低的层。在大多数系统中,无节制的 和致命的异常处理的第一帧的方法调用 堆栈(即。在更高的层次),或者一个错误信息被提交给应用程序 用户或其他一些都采取恰当的措施。注意,Hibernate也可能抛出 其他未经检查的异常,不是一个 `HibernateException`。这些 不是可收回,应采取适当的措施。

Hibernate包装 `SQLException` 年代扔与数据库进行交互 在一个 `JDBCException`。事实上,Hibernate将尝试转换异常 成一个更有意义的子类 `JDBCException`。底层 `SQLException` 总是可以通过 `JDBCException.getCause()`。Hibernate将 `SQLException` 到一个适当的 `JDBCException` 子类使用 `SQLExceptionConverter` 附加到 `SessionFactory`。默认情况下,`SQLExceptionConverter` 被定义为配置的方言。然而,它是 还可以插入自定义实现。看到的Javadocs `SQLExceptionConverterFactory` 类的细节。标准的 `JDBCException` 亚型:

- ▶ `JDBCConnectionException`:显示一个错误 与底层JDBC通信。
- ▶ `SQLGrammarException`:表示一个语法 或语法问题发出SQL。
- ▶ `ConstraintViolationException`:显示一些 形式的完整性约束违反。
- ▶ `LockAcquisitionException`:显示一个错误 获取锁级别需要执行请求的操作。
- ▶ `GenericJDBCException`:一个通用的异常 这并没有落入任何其他类别。

13 2 4. 事务超时

一个重要的功能提供了一个托管环境像EJB, 这是从来没有提供非受管代码,是事务超时。 事务 超时确保没有行为不当的事务可以无限期地绑定好 资源而没有响应返回给用户。 外一个托管(JTA) 环境,Hibernate不能完全提供此功能。 然而, Hibernate至少可以控制数据库访问操作,确保数据库 死锁和巨大的水平查询结果集定义的限制 超时。 在托管环境,Hibernate可以委托事务超时 对JTA。 这个功能是抽象的冬眠 事务 对象。

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();

    // do some work
    ...

    sess.getTransaction().commit()
}
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

setTimeout() 不能被称为CMT的豆, 在事务超时必须以声明的方式定义。

13.3. 乐观并发控制

唯一的方法,是符合高并发和高 可伸缩性是乐观并发控制和版本管理。 版本 检查使用的版本号,或时间戳,来检测冲突的更新和防止丢失更新。 Hibernate提供了三种可能的方法 来编写应用程序代码,使用开放式并发。 用例 我们讨论在上下文的长对话,但版本检查 还能防止丢失更新在单一的数据库事务。

13 3 1. 应用版本检查

在一个实现在没有多少帮助从Hibernate,每个交互 数据库出现在新 会话 和开发人员负责 重新加载所有从数据库持久化实例操纵他们之前。 应用程序是被迫进行校验,以确保自己的版本 会话事务隔离。 这种方法是最有效的方面 数据库访问。 它是最接近实体ejb方法。

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();

int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion != foo.getVersion() ) throw new StaleObjectStateException();
foo.setProperty("bar");

t.commit();
session.close();
```

这个 版本 属性映射使用 <版本>, 和Hibernate将自动增加它在冲洗如果实体 脏。

如果你是操作在低数据并发性环境,不 需要版本检查,您可以使用这种方法,跳过版本 检查。 在这种情况下, 最后提交的胜 是默认 长对话的策略。 注意,这可能 混淆用户应用程序,因为他们可能经历失去更新没有 错误消息或有机会合并冲突性的变更。

手动版本检查只是在琐碎的情况下可行 和大多数人来说不实用的应用程序。 通常不仅单一实例,但是 完整的图形对象的修改,必须检查。 Hibernate提供了自动 版本检查,不是扩展 会话 或分离实例 为设计范式。

13 3 2. 扩展的会话和自动版本

一个 会话 实例及其持久性实例 用于整个对话被称为 会话/谈话 。 Hibernate检查实例版本在冲洗时间,如果并发抛出异常 修改检测。 这是开发人员负责捕捉和处理这个异常。 常见的选项为用户的机会将变更合并或重新启动 业务对话用未过期数据。

这个 会话 断开任何底层JDBC连接吗 当等待用户交互。 这种方法是最有效的方面 的数据库访问。 应用程序不检查或版本 重新接上分离的情况下,也不需要重新加载实例在每一个 数据库事务。

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC connection, start transaction

foo.setProperty("bar");

session.flush(); // Only for last transaction in conversation
```

```
t.commit(); // Also return JDBC connection
session.close(); // Only for last transaction in conversation
```

这个 foo 对象知道这 会话 这是 载入。开始一个新的数据库事务在一个旧会话获得一个新的连接 和简历会话。提交一个数据库事务断开一个会话 从JDBC连接并返回连接池中。重新连接后,力一个版本检查数据你不更新,你可以调用 会话锁() 与 LockMode.READ 在任何对象,可能是由另一个更新 事务。 你不需要锁定任何数据 是 更新。 通常可以设置 FlushMode.MANUAL 在一个扩展的 会话,所以,只有最后一个数据库事务周期允许实际上坚持所有 在这段对话中修改。只有这最后的数据库事务 将包括 flush() 操作,然后 close() 会话结束谈话。

这种模式是有问题的 会话 太大 被存储在用户思考时间(例如,一个 HttpSession 应该 保持尽可能小)。随着 会话 也是一级缓存和包含所有加载对象,我们可以 使用这个策略只对几个请求/响应周期。使用一个 会话 只对单一的对话将很快 有陈旧的数据。

注意

早期版本的要求明确的断开和重新连接冬眠的 会话 。 这些方法已被启用,作为开始和 结束 一个事务有同样的效果。

保持断开 会话 密切 到持久层。使用有状态会话bean的EJB 保持 会话 在一个三层的环境。不要转移 web层,甚至将其序列化到一个单独的层,存储它 HttpSession 。

扩展的会话模式,或 会话/谈话 ,是 更难以实现自动当前会话上下文管理。 你需要提供自己的实现 CurrentSessionContext 对于这个。看到Hibernate Wiki的例子。

13 3 3. 分离对象和自动版本

每个交互与持久性存储发生在一个新的 会话 。然而,相同的持久化实例重用与数据库交互。应用程序操作的状态,原来在另一个分离实例加载 会话 然后reattaches他们使用 会话更新(), Session.saveOrUpdate() ,或 会话合并() 。

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been loaded already
t.commit();
session.close();
```

再次,Hibernate将检查实例版本在冲洗,抛出 如果冲突的更新发生异常。

你也可以叫 锁() 而不是 Update() , 和使用 LockMode.READ (执行版本检查和绕过所有 缓存)如果你确信没有被修改的对象。

13 3 4. 定制自动版本

你可以禁用Hibernate的自动版本增量为特定属性和 集合通过设置 乐观锁 映射属性 假 。 Hibernate将不再增量版本如果 房地产是脏的。

遗留数据库模式通常是静态的,不能修改。 或者,其他应用程序 可能访问相同的数据库,不知道如何处理版本号或 即使时间戳。在这两种情况下,版本不能依赖一个特定的列在一个表。 迫使一个版本检查用 比较了各个领域的状态在一行,但没有一个版本或时间戳属性映射,打开 乐观锁= "所有" 在 <类> 映射。这个概念上只能 如果Hibernate可以比较旧的和新的状态(即。 , 如果你使用一个长 会话 并不是每个请求的会话与分离对象)。

并发修改可以允许在实例的修改 使不重叠。如果你设置 乐观锁= "脏" 当映射 <类> ,Hibernate只会比较脏字段在冲洗。

在这两种情况下,用专用的版本/时间戳列或一个完整的/肮脏的字段 比较,Hibernate使用单一 更新 声明,适当 在 条款,每个实体执行版本检查 和更新信息。 如果你使用过渡持久性对级联回贴 与之相关的实体,Hibernate可以执行不必要的更新。 这通常不是 一个问题,但是 在更新 在数据库中触发可能 即使在执行已经改变分离实例。 你可以定制 这种行为通过设置 - before - update = " true " 在 <类> 映射,迫使Hibernate来 选择 实例来确保变化确实发生更新行之前。

13.4. 悲观锁

它并不打算,用户花太多的时间去忧虑关于锁定策略。 它通常是 足够的指定一个隔离级别的JDBC连接,然后顺其自然 数据库做所有的工作。 然而,高级用户可能希望获得 独家悲观锁或重新获得锁在开始一个新事务。

Hibernate会总是使用锁机制的数据库;它永远锁对象 在内存中。

这个 LockMode 类定义了不同的锁级别,可以是收购了 通过Hibernate。 一个锁是通过以下机制:

- » LockMode.WRITE 获得当Hibernate自动更新或者插入吗 一行。
- » LockMode.UPGRADE 可以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 可以获得明确的用户请求时使用 选择..... 对于更新NOWAIT 在Oracle。
- » LockMode.READ 是读取数据时自动获得冬眠吗 在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 代表没有锁。 切换到该所有对象 锁定模式的末尾 事务 。 对象与会话关联起来 通过调用 Update() 或 saveOrUpdate() 也开始 在这个锁定模式。

“明确的用户请求” 是表示在以下几个方面:

- » 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 会话负荷() 叫做与 升级 或 UPGRADE_NOWAIT ,请求的对象还没有加载 会话,对象加载使用 选择..... 更新 。 如果 load() 是呼吁的对象已装满 一个限制较少的锁的请求,Hibernate调用 锁() 为该对象。

会话锁() 执行版本号检查如果指定的锁 模式是 读, 升级 或 UPGRADE_NOWAIT 。 对于 升级 或 UPGRADE_NOWAIT , 选择..... 更新 是使用。

如果所请求的锁模式不支持数据库,Hibernate使用一个合适的 交替的模式而不是抛出异常。 这可以确保应用程序 便携式。

13.5. 连接释放模式

留下的遗产之一Hibernate 2. x JDBC连接管理 意味着一个 会话 将获得一个连接当它是第一 要求,然后保持连接到会话被关闭。 Hibernate 3. 介绍了x的概念模式,连接释放指示一个会话 如何处理它的JDBC连接。 下面的讨论是相关的 只有连接提供了通过一个配置 ConnectionProvider 。 用户提供的连接是本文讨论的范围之外。 不同的 发布模式确定的枚举值 org.hibernate.ConnectionReleaseMode :

- » 在近 :就是上面描述遗留的行为。 这个 Hibernate会话获得一个连接时,它首先需要执行一些JDBC访问 和维护,连接到会话是封闭的。
- » 交易后 :释放后的连接 org.hibernate事务 已经完成。
- » 在声明中 (也称为激进版本): 每个语句执行后释放连接。 这个激进的释放 如果这个声明是跳过叶开放资源关联于指定会话。 目前唯一的情况下发生这种情况是通过使用 org.hibernate.ScrollableResults 。

配置参数 hibernate连接释放模式 使用 指定哪个版本模式使用。 可能的值是:

- » 汽车 (默认):这个选择代表发布模式 返回的 org.hibernate.transaction.TransactionFactory.getDefaultReleaseMode() 法。 对于JTATransactionFactory,这 ConnectionReleaseMode返回。 在声明; JDBCTransactionFactory,这 ConnectionReleaseMode.AFTER_TRANSACTION返回。 不 改变这种默认行为视为失败由于这个设置的值 倾向于显示错误和/或无效的假设在用户代码。
- » 在近 :使用ConnectionReleaseMode.ON_CLOSE。 这个设置 是留给向后兼容的,但它的使用是不提倡的。
- » 交易后 :使用ConnectionReleaseMode.AFTER_TRANSACTION。 这个设置中不应使用JTA环境。 另请注意,使用 ConnectionReleaseMode。 交易后,如果一个会话被认为是在自动提交 模式,连接将被释放的释放模式是声明之后。
- » 在声明中 :使用ConnectionReleaseMode.AFTER_STATEMENT。 此外, 配置的 ConnectionProvider 咨询是否支持这个吗 设置(supportsAggressiveRelease())。 如果不是,释放模式是复位 到 ConnectionReleaseMode.AFTER_TRANSACTION。 此设置仅是安全的环境中 我们可以获得相同的底层JDBC连接每次您调用到 ConnectionProvider.getConnection() 或在自动提交的环境中 它并不重要,如果我们重建相同的连接。

第十四章。 拦截器和事件

表的内容

- 14.1. 拦截器
- 14.2. 事件系统
- 14.3. Hibernate声明式安全

它是有用的应用程序来应对某些发生的事件 在冬眠。 这允许实施通用 功能和扩展功能的冬眠。

14.1. 拦截器

这个 拦截器 从会话界面提供回调的 应用程序,允许应用程序进行检查和/或操作的属性 持久对象之前,保存、更新、删除或加载。 一个 可能的使用是为了跟踪审计信息。 例如,下面的 拦截器 自动设置 createTimestamp 当一个 可审计的 创建和更新吗 lastUpdateTimestamp 财产当一个 可审计的 是 更新。

你可以实现 拦截器 直接或扩展 EmptyInterceptor 。

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
```

```

import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
        Serializable id,
        Object[] state,
        String[] propertyNames,
        Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
        Serializable id,
        Object[] currentState,
        Object[] previousState,
        String[] propertyNames,
        Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
        Serializable id,
        Object[] state,
        String[] propertyNames,
        Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
        Serializable id,
        Object[] state,
        String[] propertyNames,
        Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
            System.out.println("Creations: " + creates + ", Updates: " + updates + "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }
}

```

有两种interceptors: 会话 范围和 SessionFactory 作用域。

一个 会话 指定作用域的拦截器 当一个会话被打开。

```
Session session = sf.withOptions( new AuditInterceptor() ).openSession();
```

一个 SessionFactory 作用域拦截器 配置 对象建立之前 SessionFactory 。 除非 一个会话打开显式地指定拦截器使用,提供的拦截器 将被应用到所有会话打开从那 SessionFactory 。 SessionFactory 范围 拦截器必须是线程安全的。 确保你不存储会话特有的州,因为多个 会议将使用这个拦截器可能并行。

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

14.2. 事件系统

如果你有特定的事件作出反应在你的持久层,你可以 还使用Hibernate 事件 体系结构。 事件 系统除了可以用,或作为替代,对拦截器。

许多方法 会话 接口关联到一个事件类型。 这个 全范围的定义事件类型被声明为枚举值 org.hibernate.event.spi.EventType 。 当一个请求是由一个的 这些方法,Hibernate 会话 生成一个适当的 事件并将其传递到配置的事件监听器的类型。 开箱即用的, 这些侦听器实现相同的处理中,这些方法总是造成。 然而,你可以自由实现一个定制的侦听器接口 (即。 , LoadEvent 处理注册实现吗 的 LoadEventListener 接口),在这种情况下他们 实现将负责处理任何 load() 请求 制成的 会话 。

注意

看到 [Hibernate开发者指南 关于注册 自定义事件监听器](#)。

听众应该被认为是无状态的,他们之间共享请求,和不应该保存任何 国家作为实例变量。

一个自定义侦听器实现适当的接口事件它想 过程和/或扩展一个方便的基类(或者甚至是默认的事件 侦听器使用Hibernate开箱即用的这些声明这不是final的 目的)。 这里是一个例子,一个自定义负载事件监听器:

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListener interface
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
        throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

14.3. Hibernate声明式安全

通常,声明式安全在Hibernate应用程序在一个会话facade管理 层。 Hibernate允许某种行为是permissioned通过JACC和授权 JAAS通过。 这是一个可选的功能,是建立在顶部的事件结构。

首先,您必须配置适当的事件监听器,使使用JACC 授权。 再次,看到 [Hibernate开发者指南](#) 的细节。 下面是一个例子,一个合适的 org.hibernate.integrator.spi.Integrator 实现这一目的。

```
import org.hibernate.event.service.spi.DuplicationStrategy;
import org.hibernate.event.service.spi.EventListenerRegistry;
import org.hibernate.integrator.spi.Integrator;
import org.hibernate.secure.internal.JACCPreDeleteEventListener;
import org.hibernate.secure.internal.JACCPreInsertEventListener;
import org.hibernate.secure.internal.JACCPreLoadEventListener;
import org.hibernate.secure.internal.JACCPreUpdateEventListener;
import org.hibernate.secure.internal.JACCSecurityListener;

public class JaccEventListenerIntegrator implements Integrator {

    private static final DuplicationStrategy JACC_DUPLICATION_STRATEGY = new DuplicationStrategy() {
        @Override
        public boolean areMatch(Object listener, Object original) {
            return listener.getClass().equals( original.getClass() ) &&
                JACCSecurityListener.class.isInstance( original );
        }

        @Override
        public Action getAction() {
            return Action.KEEP_ORIGINAL;
        }
    };

    @Override
    @SuppressWarnings( {"unchecked"} )
    public void integrate(
        Configuration configuration,
        SessionFactoryImplementor sessionFactory,
        SessionFactoryServiceRegistry serviceRegistry) {
        boolean isSecurityEnabled = configuration.getProperties().containsKey( AvailableSettings.JACC_EN
            if ( !isSecurityEnabled ) {
                return;
            }

        final EventListenerRegistry eventListenerRegistry = serviceRegistry.getService( EventListenerRegistr
            eventListenerRegistry.addDuplicationStrategy( JACC_DUPLICATION_STRATEGY );

        final String jaccContextId = configuration.getProperty( Environment.JACC_CONTEXTID );
        eventListenerRegistry.prependListeners( EventType.PRE_DELETE, new JACCPreDeleteEventListener(j
        eventListenerRegistry.prependListeners( EventType.PRE_INSERT, new JACCPreInsertEventListener(ja
        eventListenerRegistry.prependListeners( EventType.PRE_UPDATE, new JACCPreUpdateEventListener
        eventListenerRegistry.prependListeners( EventType.PRE_LOAD, new JACCPreLoadEventListener(jacc

    }
}
```

你还必须决定如何配置您的JACC提供程序。 一个选项是告诉Hibernate什么权限 绑定到什么角色,它配置JACC提供程序。 这

将是完成 hibernate cfg xml 文件。

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions="*/>
```

第十五章。批处理

表的内容

- 15.1. 批量插入
- 15.2. 批量更新
- 15.3. StatelessSession接口的
- 15.4. dml风格操作

一个天真的方法在数据库中插入100000行使用Hibernate的可能 看起来像这样:

```
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();
```

这将摔倒的 OutOfMemoryException 某处 在第50000行。 这是因为Hibernate缓存所有新插入的 客户 在会话级缓存实例。 在这一章 我们将向您展示如何避免这个问题。

如果您在进行批处理你需要支持使用 JDBC批处理。 这是绝对必要的,如果你想达到最优性能。 设置JDBC批处理大小数量合理(10 - 50为例):

```
hibernate.jdbc.batch_size 20
```

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。

你也可以做这类工作的过程,互动 第二级缓存是完全禁用:

```
hibernate.cache.use_second_level_cache false
```

然而,这并不是绝对必要的,因为我们可以显式地设置 CacheMode 禁用交互与二级缓存。

15.1. 批量插入

当进行新的对象持久化 flush() 和 然后 clear() 会话经常为了控制大小 第一级缓存。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

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

15.2. 批量更新

用于检索和更新数据,同样的想法应用。 此外,你需要 使用 滚动() 利用服务器端游标为 查询,该查询返回多行数据。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
```



```

        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();

```

15.3. StatelessSession接口的

另外,Hibernate提供了一个API,可以面向命令用于 流数据与数据库分离对象的形式。 一个 StatelessSession 没有持久化上下文有关 ,不提供许多高级生命周期的语义。 特别是,一个无状态会话没有实现一级缓存也没有 与任何二级或查询缓存。 它没有实现 事务写入后或自动脏检查。 操作 使用无状态会话从来没有级联到关联的实例。 集合 都忽略了一个无状态会话。 通过一个无状态会话操作 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。

```

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();

```

在这个代码示例中, 客户 实例返回 查询立即分离。 他们从来没有产生任何持久性 上下文。

这个 插入()、update() 和 删除() 操作 定义的 StatelessSession 界面被认为是 直接数据库行级操作。 他们导致立即执行的 SQL 插入、更新 或 删除 分别。 他们有不同的语义 save(),saveOrUpdate() 和 删除() 操作定义的 会话 接口。

15.4. dml风格操作

正如已经讨论过的,自动和透明的对象/关系映射而言 与管理对象的状态。 对象状态可在记忆。 这意味着直接在数据库中操作数据(使用SQL 数据操纵语言 (DML)声明: 插入,更新,删除) 不会影响内存中的状态。 然而,Hibernate提供了方法 对于散装 sql风格的DML语句执行,执行是通过的 Hibernate查询语言(第十六章, HQL:Hibernate查询语言)。

这个伪语法为 更新 和 删除 语句 是: (更新|删除)? EntityName(情况)? 。

一些注意事项:

- ▶ 在FROM子句,从关键字是可选的
- ▶ 只能有一个单一的实体命名在from子句。 它能,然而,是 别名。 如果实体名称别名,那么任何属性引用必须 有资格使用该别名。 如果实体的名字不是别名,那么它就是 违法的任何属性引用是合格的。
- ▶ 没有 16.4节,“形式的连接语法” ,隐式或显式, 可以指定在一个散装HQL查询。 可以使用子查询的where子句,在那里 这个子查询本身可能包含连接。
- ▶ where子句的也是可选的。

作为一个例子,执行一个HQL 更新 ,使用 Query.executeUpdate() 法。 该方法被命名为 那些熟悉JDBC的 PreparedStatement.executeUpdate() :

```

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

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

tx.commit();
session.close();

```

为了与EJB3规范,HQL 更新 语句,默认情况下,不影响 部分5 1 3 1,“版本号” 或 部分5 1 3 2,“时间戳” 属性值 受影响的实体。 然而, 你可以迫使Hibernate重置 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。

```

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

tx.commit();

```

```
session.close();
```

定制版本类型, `org.hibernate.usertype.UserVersionType`, 不允许联同一个吗 更新版本 语句。

执行一个HQL 删除,使用相同的 `Query.executeUpdate()` 方法:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
    .setString( "oldName", oldName )
    .executeUpdate();
tx.commit();
session.close();
```

这个 `int` 返回的值 `Query.executeUpdate()` 法显示实体的数量影响操作。 这可能会或可能不会 相互关联影响的行数在数据库中。 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 号表明实际实体的数量影响的语句。 回到 例子,加入子类,子类的删除与实际上可能结果 在删除对不仅仅是表的子类映射,同时也是“根” 表和潜在的更低的加入子类表继承层次结构。

这个伪语法为 插入 语句是: 插入EntityName属性列表中选择语句 。 一些 注意事项:

- ▶ 只有插入..... 选择..... 形式是支持,而不是插入..... 值..... 形式。

这个属性列表是类似的 列规范 在SQL 插入 语句。 对于实体参与映射 继承,只有属性直接定义在给定类别可以 用于属性列表。 超类和子类的属性都是不允许的 属性没有意义。 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,然而,导致问题在冬眠 类型 年代, 等效 而不是 平等。 这可能导致 问题与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 数据库可能不区别或也许能够处理转换。
- ▶ 为id属性,insert语句给你两个选择。 你可以 显式地指定id属性在属性列表中,在这种情况下,它的价值 来自相应的选择表达,或省略的属性列表, 在这种情况下生成的值是使用。 这后一个选项只有当 使用id发电机运行在数据库;试图使用这个选项 任何“记忆”式发电机将导致异常在解析。 这次讨论的目的,数据库内发电机被认为是 `org.hibernate.id.SequenceGenerator` (和它的子类)和 任何的实现者 `org.hibernate.id.PostInsertIdentifierGenerator`。 这里最值得注意的例外 `org.hibernate.id.TableHiLoGenerator`, 这不能被使用,因为它没有暴露一个可选择的方式获得其值。
- ▶ 对于属性映射为要么 版本 或 时间戳, insert语句给你两个选择。 您可以指定属性 属性列表,在这种情况下,它的价值来自于相应选择表达式, 或省略的属性列表,在这种情况下 种子值 定义的 `org.hibernate.type.VersionType` 是使用。

以下是一个例子,一个HQL 插入 语句执行:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

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

第十六章。 HQL:Hibernate查询语言

表的内容

- 16.1. 大小写敏感性
- 16.2. 从条款的
- 16.3. 协会和连接
- 16.4. 形式的连接语法
- 16.5. 标识符属性
- 16.6. `select`子句
- 16.7. 聚合函数
- 16.8. 多态查询
- 16.9. `where`子句
- 16.10. 表达式
- 16.11. `order by`子句
- 16.12. 该集团通过条款
- 16.13. 子查询
- 16.14. HQL的例子
- 16.15. 批量更新和删除
- 16.16. 提示和技巧
- 16.17. 组件
- 16.18. 行值的构造函数的语法

Hibernate使用强大的查询语言(HQL)类似的样子 `sql`。 与SQL,然而,是完全面向对象的HQL 和理解的理念如继承、多态性和协会。

16.1. 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` 不是 `org.hibernate如foo` ,和 `foo.barSet` 不是 `foo.BARSET` 。

本手册使用小写HQL关键词。一些用户发现查询与大写的关键词 更可读的,但本公约不适合查询嵌入的Java代码。

16.2. 从条款的

最简单的可能的Hibernate查询的表单:

```
from eg.Cat
```

这将返回类的所有实例 如猫 。 你通常不需要限定类名,因为 `汽车进口` 是默认的。 例如:

```
from Cat
```

以参考 `猫` 其他地区的 查询,你需要分配一个 别名 。 例如:

```
from Cat as cat
```

这个查询指定别名 `猫` 到 `猫` 实例,那么您可以使用别名查询中,稍后。 这个 `作为` 关键字是可选的。 你也可以写:

```
from Cat cat
```

多个类可以出现,导致一个笛卡儿积或 “十字” 加入。

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

这是很好的实践名称查询别名使用一个初始小写,这是 符合标准的Java命名为局部变量 (如 `domesticCat`)。

16.3. 协会和连接

您还可以指定别名关联实体或元素 值集合使用 `加入` 。 例如:

```
from Cat as cat
inner join cat.mate as mate
left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

支持的连接类型是借用ANSI SQL:

- » 内加入
- » 左外连接
- » 右外连接
- » 全部加入 (不是通常有用)

这个 `内加入` , `左外连接` 和 `右外连接` 构造可能是缩写。

```
from Cat as cat
join cat.mate as mate
left join cat.kittens as kitten
```

你还可以提供额外的加入条件使用HQL 与 `关键字`。

```
from Cat as cat
left join cat.kittens as kitten
with kitten.bodyWeight > 10.0
```

“获取”加入允许协会或值的集合 初始化以及它们的父对象使用一个单一的选择。 这是特别 有用的对于一个集合。 它有效地覆盖了外部联接和 懒惰的声明映射文件和集合的关联。 看到 [20.1节](#), “[抓取策略](#)” 为更多的信息。

```
from Cat as cat
inner join fetch cat.mate
left join fetch cat.kittens
```

一个获取加入通常不需要分配一个别名,因为相关的对象 不应该用于 在 条款(或任何其他条款)。 关联的对象也没有直接返回

的查询结果。相反,他们可能通过父对象被访问。唯一的原因,你可能需要一个别名是如果你递归地连接抓取一个进一步的收集:

```
from Cat as cat
  inner join fetch cat.mate
  left join fetch cat.kittens child
  left join fetch child.kittens
```

这个取构造不能用于查询称为使用迭代() (尽管滚动()可以使用)。取应该一起使用 setMaxResults() 或 setFirstResult() ,因为这些操作都是基于结果的行通常包含副本渴望收集获取,因此,行数不是你所期望的那样。取应该也不是一起使用与即兴与条件。它可以创建一个笛卡儿积的连接抓取超过一个集合在一个查询,所以照顾在这种情况下。加入收藏获取多个角色可以生产意外结果袋的映射,所以用户谨慎建议制定查询在这个时候案例。最后,请注意,全部加入取和正确连接获取是没有意义的。

如果您使用的是属性级惰性抓取(字节码插装),它是可能迫使Hibernate来获取延迟属性在第一个查询立即使用获取所有属性。

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

16.4. 形式的连接语法

HQL支持两种形式的协会加入: 隐式和明确。

查询前一节所示所有使用明确形式,即连接关键字显式地使用在from子句中。这是推荐的形式。

这个隐式形式不使用join关键字。相反,协会是“反向”使用一样。隐式加入可以出现在任何HQL条款。隐式连接结果在内部连接在生成的SQL语句。

```
from Cat as cat where cat.mate.name like '%s%'
```

16.5. 标识符属性

有2种方法可以指一个实体的标识符属性:

- ▶ 特殊属性(小写) id 可以用来引用标识符属性,一个实体如果实体没有定义一个非标识符属性命名为id。
- ▶ 如果实体定义一个命名标识符属性,您可以使用这个属性名。

引用标识符属性组合遵循相同的命名规则。如果实体有一个非标识符属性命名为id,复合标识符属性只能通过它定义引用命名。否则,特别 id 财产可以用来引用标识符属性。

重要

请注意,开始在版本3.2.2,这已经发生了很大的改变。在以前的版本中, id 总是提到了标识符属性,不管实际的名字。一个分枝的决定是,非标识符属性命名 id 永远不可能被引用在Hibernate查询。

16.6. select子句

这个选择条款选择哪些对象和属性返回查询的结果集。考虑以下:

```
select mate
from Cat as cat
  inner join cat.mate as mate
```

查询将选择伴侣年代的其他猫年代。你可以表达这个查询更简洁:

```
select cat.mate from Cat cat
```

查询可以返回属性的任何值类型包括属性的组件类型:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

查询可以返回多个对象和/或属性数组的类型 Object[] :

```
select mother, offspr, mate.name
from DomesticCat as mother
```

```
inner join mother.mate as mate
left outer join mother.kittens as offspr
```

或作为一个 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
```

或者,假设类 家庭 有一个适当的构造函数——作为一个实际的类型安全的Java对象:

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

你可以指定别名来选定的表达式使用 作为 :

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

这是最有用的,当配合使用 选择新地图 :

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

这个查询返回一个 地图 从选择别名值。

16.7. 聚合函数

HQL查询甚至可以返回结果聚合函数的性质:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

支持的聚合函数是:

- » avg(...),金额(...),min(...),马克斯(...)
- » count(*)
- » 数(...),计数(不同的...),计数(所有.....)

您可以使用算术运算符,连接,和公认的SQL函数 在select子句:

```
select cat.weight + sum(kitten.weight)
from Cat cat
join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

这个 截然不同的 和 所有 可以使用关键字和 有相同的语义在SQL。

```
select distinct cat.name from Cat cat
select count(distinct cat.name), count(cat) from Cat cat
```

16.8. 多态查询

查询:

```
from Cat as cat
```

返回实例不仅 猫 ,但也喜欢子类 domesticCat 。 Hibernate查询名称 任何 Java 类或接口的 从 条款。 该查询将返回实例所有的持久化类,扩展这个类或实现接口。 以下 查询将返回所有持久化对象:

```
from java.lang.Object o
```

接口 命名 可能实现的各种持久 类:

```
from Named n, Named m where n.name = m.name
```

最后这两个查询需要不止一个SQL 选择 。 这 意味着 订单 条款不正确订单的整个结果集。 这也意味着你不能叫这些查询使用 查询滚动() 。

16.9. where子句

这个在子句可以提炼的实例列表返回。如果没有别名存在,您可以参考属性的名字:

```
from Cat where name='Fritz'
```

如果有一个别名,使用合格的属性名称:

```
from Cat as cat where cat.name='Fritz'
```

这返回的实例 猫 名为“弗里茨”。

下面的查询:

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

返回的所有实例 foo 与一个实例的酒吧与日期财产等于 startDate可以财产的 foo。复合路径表达式使在条款非常强大。考虑以下:

```
from Cat cat where cat.mate.name is not null
```

这个查询转换成一个SQL查询与一个表(内部)加入。例如:

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

会导致查询,需要四个表连接在SQL。

这个 = 操作符可以用来比较不仅属性,但也实例:

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

特殊属性(小写) id 可以用来参考的吗一个对象的唯一标识符。看到 16.5节,“标识符属性”为更多的信息。

```
from Cat as cat where cat.id = 123
from Cat as cat where cat.mate.id = 69
```

第二个查询是高效的和不需要一个表连接。

复合材料的性能标识符也可以被使用。考虑下面的例子人有复合标识符组成的吗国家和 medicareNumber:

```
from bank.Person person
where person.id.country = 'AU'
and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
and account.owner.id.medicareNumber = 123456
```

再次,第二个查询不需要表连接。

看到 16.5节,“标识符属性”为更多的信息关于引用标识符属性)

特殊的财产类访问discriminator值的一个实例的情况多态的持久性。一个Java类名中嵌入的 where子句将翻译其鉴别器值。

```
from Cat cat where cat.class = DomesticCat
```

您还可以使用组件或复合用户类型或属性的表示组件类型。看到 16.17节,“组件”为更多的信息。

一个“任何”类型的特殊属性 id 和类这允许您表达一个加入以下方式(在那里 AuditLog.item 是一个属性映射 <任何>):

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

这个日志条目类和付款类将参考价值的完全不同的数据库列在上面的查询。

16.10. 表达式

表达式用于在条款包括以下:

- 数学运算符: +,-,*,/
- 二进制比较运算符: =, > =, < =, < >, !=,像
- 逻辑操作 和,或者,不
- 括号 () 这表明分组
- 在, 不是在, 之间, 是NULL, 不是空, 是空的, 不是空, 成员 和 不是会员的
- “简单”的情况下, 案例..... 当..... 然后..... 别的... 结束, 和 “搜索”的情况下, 情况下, 当..... 然后..... 别的... 结束
- 字符串连接 ...|... 或 concat(.....)
- 当前日期(), 当前时间(), 和 当前时间戳的值()
- 第二(...), 分钟(...), 小时(...), 天(...), 月(...), 和 年(...)
- 任何函数或操作符定义为ejb ql 3.0:
substring(), trim(), 较低的(), 上层(), 长度(), 定位(), abs(), sqrt(), 钻头长度(), 国防部()
- 合并() 和 nullif()
- str() 转换为数字或时间值 可读的字符串
- 铸造(... 为...), 第二个参数是这个名字的一个Hibernate类型和 提取(... 从...) 如果ANSI 铸() 和 提取物() 支持 底层数据库
- HQL的 指数() 函数, 适用于的别名 一个加入索引收集
- HQL函数, 把收藏值的路径表达式: 大小(), minelement(), maxelement maxindex minindex(), 随着 特殊 元素() 和 指数 功能 可以量化的使用 其中的一些、全部存在, 任何在。
- 任何支持的SQL标量函数像 符号(), trunc(), rtrim(), 和 罪()
- jdbc风格位置参数 吗?
- 命名参数 :名字, :开始日期, 和 :x1
- SQL文字 “foo”, 69年, 6.66 e + 2, “1970-01-01 10:00:01.0”
- Java 公共静态最终 常量 如色虎斑

在 和 之间 可以用如下:

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

否定的形式可以写成如下:

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

同样的, 是NULL 和 不是空 可以用来测试吗 null值。

布尔值可以很容易地用于表达式通过声明HQL查询替换在冬眠 配置:

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

这将替换关键字 真正的 和 假 与 文字 1 和 0 在翻译的SQL从这个HQL:

```
from Cat cat where cat.alive = true
```

你可以测试集合的大小与特殊的属性 大小 或 特殊的 大小() 函数。

```
from Cat cat where cat.kittens.size > 0
```

```
from Cat cat where size(cat.kittens) > 0
```

对于索引的集合, 你可以参考最小和最大指标使用 minindex 和 maxindex 功能。 同样的, 你可以参考最小和最大元素集合的基本类型 使用 minelement 和 maxelement 功能。 例如:

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

```
from Order order where maxindex(order.items) > 100
```

```
from Order order where minelement(order.items) > 10000
```

SQL函数 任何, 一些, 所有的, 存在, 在 当通过元素的支持 或指数组集合(元素 和 指数 函数) 或子查询的结果(见下图):

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

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

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

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

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

注意,这些构造- 大小, 元素, 指数, minindex, maxindex, minelement, maxelement ——只能用于 在Hibernate3 where子句。

元素的索引集合(数组,列表和地图)参考 指数在where子句只有:

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

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

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

里面的表达式 [] 甚至可以是一个算术表达式:

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL还提供内置的 指数() 功能元素 一个一对多的协会或集合值。

```
select item, index(item) from Order order
join order.items item
where index(item) < 5
```

标量SQL函数由底层数据库可以使用:

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

考虑多少时间和更少的可读的 以下查询将在SQL:

```
select cust
from Product prod,
Store store
inner join store.customers cust
where prod.name = 'widget'
and store.location.name in ( 'Melbourne', 'Sydney' )
and prod = all elements(cust.currentOrder.lineItems)
```

提示: 类似

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
stores store,
locations loc,
store_customers sc,
product prod
WHERE prod.name = 'widget'
AND store.loc_id = loc.id
AND loc.name IN ( 'Melbourne', 'Sydney' )
AND sc.store_id = store.id
AND sc.cust_id = cust.id
AND prod.id = ALL(
SELECT item.prod_id
FROM line_items item, orders o
WHERE item.order_id = o.id
AND cust.current_order = o.id
)
```

16.11. order by子句

返回的列表中查询可以订购任何属性返回类或组件:

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

可选的 asc 或 Desc 表示升序或降序排列 分别。

16.12. 该集团通过条款

一个查询,该查询返回聚合值可以按任何属性返回的类或组件:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```



```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

一个有 条款也是允许的。

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

SQL函数和聚合函数是允许的 有 和 订单 如果他们支持的条款基础数据库 (即。 ,而不是在MySQL)。

```
select cat
from Cat cat
join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

无论是 集团 条款和 订单 条款可以包含算术表达式。 Hibernate也并不目前扩大一个分组的实体, 所以你不能写集团通过猫 如果所有的属性的 猫 正非聚合。 你必须列出所有 非聚合属性显式。

16.13. 子查询

对于数据库,支持subselects,Hibernate支持子查询在查询。 子查询必须 身边都是括号(通常由一个SQL聚合函数调用)。 即使相关子查询 (子查询,指一个别名在外层查询)是允许的。

```
from Cat as fatcat
where fatcat.weight > (
  select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
  select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
  from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
  select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

注意,HQL子查询可以只发生在选择或条款。

注意,子查询也可以利用 行值的构造函数 语法。 看到 16.18节,“[行值的构造函数语法](#)” 为更多的信息。

16.14. HQL的例子

Hibernate查询可以很强大的和复杂的。 事实上,权力的查询语言 是Hibernate的主要优势。 下面的示例查询类似于查询 这已经用最近的工程。 请注意,大多数查询你会写会比下面的例子更简单。

下面的查询返回订单id,项的数量,给定的最小值和总订单的总价值为所有 未支付的订单为一个特定的客户。 结果所安排 总价值。 在确定价格时,它使用当前目录。 生成的SQL查询,反对 秩序, ORDER_LINE, 产品, 目录 和 价格 表有四个内部连接和一个 (不相关的)subselect。

```
select order.id, sum(price.amount), count(item)
from Order as order
join order.lineItems as item
join item.product as product,
Catalog as catalog
join catalog.prices as price
where order.paid = false
and order.customer = :customer
and price.product = product
and catalog.effectiveDate < sysdate
and catalog.effectiveDate >= all (
  select cat.effectiveDate
  from Catalog as cat
  where cat.effectiveDate < sysdate
)
group by order
```

```
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

什么一个怪物! 其实,在现实生活中,我不是很喜欢子查询,所以我查询 真的更像这个:

```
select order.id, sum(price.amount), count(item)
from Order as order
  join order.lineItems as item
  join item.product as product,
  Catalog as catalog
  join catalog.prices as price
where order.paid = false
  and order.customer = :customer
  and price.product = product
  and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

接下来的查询项支付的数量在每个状态,排除所有支付的 待批 状态,最近的状态变化是由 当前用户。 它转换成一个SQL查询和两个内部连接和相关subselect 反对 付款, 付款状态 和 付款状态变化 表。

```
select count(payment), status.name
from Payment as payment
  join payment.currentStatus as status
  join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
  or (
    statusChange.timeStamp = (
      select max(change.timeStamp)
      from PaymentStatusChange change
      where change.payment = payment
    )
    and statusChange.user <> :currentUser
  )
group by status.name, status.sortOrder
order by status.sortOrder
```

如果 statusChanges 是作为一个列表集合映射,而不是一套, 查询将会更简单的写。

```
select count(payment), status.name
from Payment as payment
  join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
  or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

接下来的查询使用MS SQL Server isNull() 函数来返回所有 账户和无偿支付的单位当前用户所属。 它转换成一个SQL查询三个内部连接,外连接和subselect反对 这个 帐户, 付款, 付款状态, 帐户类型, 组织 和 org用户 表。

```
select account, payment
from Account as account
  left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
  and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

对于一些数据库,我们需要废除(相关)subselect.

```
select account, payment
from Account as account
  join account.holder.users as user
  left outer join account.payments as payment
where :currentUser = user
  and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

16.15. 批量更新和删除

HQL现在支持 更新, 删除 和 插入..... 选择..... 语句。 看到 15.4节, “dml风格操作” 为更多的信息。

16.16. 提示和技巧

你可以数一数没有返回他们的查询结果:

```
((Integer) session.createQuery("select count(*) from ...").iterate().next()).intValue()
```

订购一个结果集的大小,请使用以下查询:

```
select usr.id, usr.name
```

```
from User as usr
  left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

如果你subselects数据库支持,您可以将一个条件在选择 尺寸在where子句的查询:

```
from User usr where size(usr.messages) >= 1
```

如果您的数据库不支持subselects,使用以下查询:

```
select usr.id, usr.name
from User usr
  join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

由于该解决方案不能返回 用户 与零信息 因为内心的加入,以下表格也很有用:

```
select usr.id, usr.name
from User as usr
  left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

属性的一个JavaBean可以绑定到命名查询参数:

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

集合是可分页使用 查询 接口与一个过滤器:

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

集合的元素可以命令或分组使用查询过滤器:

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

你可以找到一个集合的大小没有初始化:

```
( (Integer) session.createQuery("select count(*) from ...").iterate().next() ).intValue();
```

16.17. 组件

组件可以使用类似于简单的值类型,用来在HQL 查询。 他们可以出现在 选择 条款如下:

```
select p.name from Person p
```

```
select p.name.first from Person p
```

在这个人的名字属性是一个组件。 组件也可以被使用 在 条款:

```
from Person p where p.name = :name
```

```
from Person p where p.name.first = :firstName
```

组件也可以使用的 订单 条款:

```
from Person p order by p.name
```

```
from Person p order by p.name.first
```

另一个普遍使用的组件 [16.18节](#), “[行值的构造函数语法](#)” 。

16.18. 行值的构造函数的语法

HQL支持使用ANSI SQL 行值的构造函数 语法,有时 称为 tuple 语法,即使底层数据库可能不支持 这一观念。 在这里,我们通常是指多值比较,通常相关 与组件。 考虑一个实体定义一个人的名字成分:

```
from Person p where p.name.first='John' and p.name.last='Jingleheimer-Schmidt'
```

这是有效的语法虽然有点冗长。 你可以使它更加简洁的使用 行值的构造函数 语法:

```
from Person p where p.name=('John', 'Jingleheimer-Schmidt')
```

它还可以指定这个的 选择 条款:

```
select p.name from Person p
```

使用 行值的构造函数 语法也可以是有益的 当使用子查询,需要比较多个值:

```
from Cat as cat
where not ( catname, cat.color ) in (
  select cat.name, cat.color from DomesticCat cat
)
```

有一件事要考虑如果你想使用这个语法,是查询 被依赖的排序组件子属性的元数据。

第十七章。 标准查询

表的内容

- 17.1. 创建一个 标准 实例
- 17.2. 缩小结果集
- 17.3. 排序结果
- 17.4. 协会
- 17.5. 动态关联获取
- 17.6. 组件
- 17.7. 集合
- 17.8. 示例查询
- 17.9. 预测、聚合和分组
- 17.10. 超然的查询和子查询
- 17.11. 查询通过自然标识符

Hibernate的特色是直观的、可扩展的标准查询API。

17.1. 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();
```

17.2. 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();
```

17.3. 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();
```

17.4. 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。

17.5. 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 [20.1节,“抓取策略”](#) 为更多的信息。

17.6. 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 [17.7节,“集合”](#)

17.7. 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 [17.4节,“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();
```

17.8. 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();
```

17.9. 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")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .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();
```

17.10. 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();
```

17.11. 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();
```


第18章。原生SQL

表的内容

18.1. 使用 SqlQuery

- 18 1 1. 标量查询
- 18 1 2. 实体查询
- 18 1 3. 处理协会和集合
- 18 1 4. 返回多个实体
- 18 1 5. 返回非受管实体
- 18 1 6. 处理继承
- 18 1 7. 参数

18.2. 命名的SQL查询

- 18 2 1. 使用返回属性来显式地指定列/别名 名称
- 18 2 2. 使用存储过程查询

18.3. 自定义的SQL创建、更新和删除

18.4. 定制SQL加载

你也可以表达查询本机SQL方言的 数据库。这是有用的,如果你想利用特定于数据库的特性 如查询提示或 连接 关键字在甲骨文。它 还提供了一个干净的迁移路径从一个直接的SQL / JDBC的基础 Hibernate应用程序。

Hibernate3允许您指定手写的SQL,包括存储 程序,因为所有的创建、更新、删除和加载操作。

18.1. 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.createQuery()。以下部分 描述如何使用这个API 查询。

18 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.createQuery("SELECT * FROM CATS").list();
sess.createQuery("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 ResultSetMeta Data ,或者只是更明确的在 返回的是什么,一个人可以用吗 addScalar() :

```
sess.createQuery("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 ResultSetMeta Data 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 * 和可能的回报率高于3家上市 列。

它可以保留全部或部分类型信息 标量的。

```
sess.createQuery("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 ResultSetMeta Data 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ResultSetMeta Data 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。

18 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。

18 1 3. 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 WHERE
.addEntity("cat", Cat.class)
.addJoin("cat.dog");
```

在本例中,返回的猫's将他们的狗属性完全初始化没有任何额外的往返数据库。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/列名是不够的。

18 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。

列别名注入需要以下查询(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()
```

18 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18.1. 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。 * }
一个收集键	{{aliasname}}键}	ORGID为{胶原键}
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。 * }

18 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。

18 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。

18 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();
```

18.2. 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 节11 4 1 7, “外化命名查询”)。 在这种情况下,你做的 不需要调用 addEntity() 。

例18.1. 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18.2. 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 <加载收集> 元素是 用于定义查询的初始化集合,

例18.3. 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18.4. 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18.5. < 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18.6. 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 annotations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18.7. “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`。 `resultSetMapping`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18.8, “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18.9, “使用点符号在@FieldResult指定关联”](#)。

例18.7. 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18.8. 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18.9。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 surface",
    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", resultClass=
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18.10. 标量值通过 @ColumnResult

```

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

```

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。

18 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>
</sql-query>

```

```

</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 {} 语法对于注入。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 <返回鉴别器> 指定 鉴别器列。

18 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 <加载收集> 不支持。

18 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。

18.3. 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 5.6节,“柱变形金刚:阅读和写作表达” 。例18.11,“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18.11. 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 CHAOS")
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18.12, “定制CRUD XML”](#) 。

例18.12. 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 xml:id=?</sql-update>
  <sql-delete>DELETE FROM PERSON WHERE xml: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annotations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18.13, “重写SQL语句的集合使用 注释”](#) 。

例18.13. 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18.14. 重写SQL语句对于二次表

```
@Entity
@SecondaryTables({
    @SecondaryTable(name = "Cat nbr1"),
})
```

```

@SecondaryTable(name = "Cat2"))
@org.hibernate.annotations.Tables( {
    @Table(applyTo = "Cat", comment = "My cat table" ),
    @Table(applyTo = "Cat2", foreignKey = @ForeignKey(name="FK_CAT2_CAT"), fetch = FetchType.SELECT,
        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 反乌的操作:

例18.15. 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;

```

18.4. 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 5.6节,“柱变形金刚:阅读和写作表达” 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 xml: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 xml:id=?
</sql-query>
```

注释等效 <装载机> 是 @Loader注释见 例18.11, “定制CRUD通过注释” 。

第十九章。 过滤数据

表的内容

19.1. Hibernate过滤器

提供了一个创新的新方法Hibernate3来处理数据 “可见性” 规则。 一个 Hibernate过滤器 是一个全球性的, 命名、参数化过滤器,可以启用或禁用特定 Hibernate会话。

19.1. Hibernate过滤器

Hibernate3有能力预先过滤条件和附加 这些过滤器两类级别和水平集合。 一个过滤器 标准允许您定义一个限制条款类似于现有的 “哪里” 属性可以在类和各种集合的元素。 这些过滤条件,然而,可以参数化的。 应用程序 可以在运行时再决定是否某些过滤器应该启用并吗 他们的参数值应该。 过滤器可用于像数据库 的观点,但他们是参数化的应用程序的内部。

使用annotatons定义过滤器通过 @org.hibernate.annotations.FilterDef 或 @org.hibernate.annotations.FilterDefs 。 一个过滤器 定义有一个 名称() 和一个数组的 参数()。 一个参数将允许您调整的行为 过滤器在运行时。 每个参数所定义的 @ParamDef 这有一个名称和一个类型。 你也可以 定义一个 defaultCondition() 参数对于一个给定的 @FilterDef 设置默认条件时使用 没有定义在每个个体 @Filter 。 @FilterDef (年代)可以被定义在类或包 水平。

我们现在需要定义SQL过滤器条款应用于要么 实体加载或收集负载。 @Filter 使用和 无论是在实体或放置集合元素。 连接之间 @FilterName 和 @Filter 是一个匹配的名称。

例19.1. @FilterDef和@Filter注释

```
@Entity
@FilterDef(name="minLength", parameters=@ParamDef( name="minLength", type="integer" ))
@Filters( {
  @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length"),
  @Filter(name="minLength", condition=":minLength <= length")
})
public class Forest { ... }
```

当使用一个协会收集表作为一个关系 表示,您可能想要应用的筛选条件 协会表本身或目标实体表。 应用 约束目标实体,使用常规 @Filter 注释。 然而,如果你想要的目标 关联表,使用 @FilterJoinTable 注释。

例19.2. 使用 @FilterJoinTable 对于filterting在 关联表

```
@OneToMany
@JoinTable
//filter on the target entity table
@Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length")
//filter on the association table
@FilterJoinTable(name="security", condition=":userlevel >= requiredLevel")
public Set<Forest> getForests() { ... }
```

使用Hibernate映射文件来定义过滤器形势是 非常类似的。 过滤器必须首先定义,然后附加到 适当的映射元素。 定义一个过滤器,使用 <过滤器def / > 元素在一个 < hibernate映射/ > 元素:

例19.3. 定义一个过滤器定义通过 <过滤器def >

```
<filter-def name="myFilter">
  <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

这个过滤器可以被附加到一个类或集合(或者, 两个或多个胞胎每同时):

例19.4. 附加一个过滤器,一个类或集合使用 <过滤器>

```
<class name="myClass" ...>
  ...
  <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>

  <set ...>
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
  </set>
</class>
```

这个方法 会话 是: `enableFilter(String filterName)` , `getEnabledFilter(String filterName)` 和 `disableFilter(String filterName)` 。 默认情况下,过滤器 是 不 使给定的会话。 过滤器必须 通过使用 `Session.enableFilter()` 方法,该方法返回的一个实例 过滤器 接口。 如果您使用上面定义简单的过滤器,它看起来像 这个:

```
session.enableFilter("myFilter").setParameter("myFilterParam", "some-value");
```

方法在org.hibernate。 过滤界面做的允许 常见的方法链接太多Hibernate。

以下是一个完整的示例,使用时态数据的 有效的记录日期模式:

```
<filter-def name="effectiveDate">
  <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
  ...
  <many-to-one name="department" column="dept_id" class="Department"/>
  <property name="effectiveStartDate" type="date" column="eff_start_dt"/>
  <property name="effectiveEndDate" type="date" column="eff_end_dt"/>
  ...
  <!--
    Note that this assumes non-terminal records have an eff_end_dt set to
    a max db date for simplicity-sake
  -->
  <filter name="effectiveDate"
    condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
  ...
  <set name="employees" lazy="true">
    <key column="dept_id"/>
    <one-to-many class="Employee"/>
    <filter name="effectiveDate"
      condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
  </set>
</class>
```

为了确保您提供目前有效 记录,使过滤器在会话之前,检索员工 数据:

```
Session session = ...;
session.enableFilter("effectiveDate").setParameter("asOfDate", new Date());
List results = session.createQuery("from Employee as e where e.salary > :targetSalary")
    .setLong("targetSalary", new Long(1000000))
    .list();
```

尽管薪水约束是明确的提到 结果在上面的HQL,因为启用过滤器、查询 只返回当前活跃的员工薪水大于1 百万美元。

如果你想使用过滤器和外连接,通过HQL或 负载抓取,小心的条件表达式的方向。 它是安全的进行这个设置为左外连接。 把参数首先 其次是列名(s)后的运营商。

在被定义,一个过滤器可能会被附加到多个实体 和/或集合每个都有它自己的条件。 这是有问题的 当条件是相同的每一次。 使用 <过滤器def /> 允许你define默认 条件,无论是作为一个属性或CDATA:

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-def>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

这将使用默认的条件只要过滤器连接 到的东西,而没有指定一个条件。 这意味着你可以给一个 特定的条件作为附件的过滤器,覆盖 默认条件下,在特定的情况下。

第二十章. 提高性能

表的内容

20.1. 抓取策略

- 20 1 1. 处理懒惰协会
- 20 1 2. 调谐获取策略
- 20 1 3. 单端协会代理
- 20 1 4. 初始化集合和代理
- 20 1 5. 使用批量抓取
- 20 1 6. 使用subselect抓取
- 20 1 7. 获取配置文件
- 20 1 8. 使用延迟属性获取

20.2. 二级缓存

- 20 2 1. 缓存映射
- 20 2 2. 策略:只读
- 20 2 3. 策略:读/写
- 20 2 4. 策略:nonstrict读/写
- 20 2 5. 策略:事务
- 20 2 6. 缓存提供者/并发策略的兼容性

20.3. 管理缓存

20.4. 查询缓存

- 20 4 1. 启用查询缓存
- 20 4 2. 查询缓存区域

20.5. 理解集合的性能

- 20 5 1. 分类法
- 20 5 2. 列表,地图,idbags和集是最有效的集合 更新
- 20 5 3. 袋和列表是最有效的逆集合
- 20 5 4. 一次删除

20.6. 监控性能

- 20 6 1. 监测SessionFactory
- 20 6 2. 指标

20.1. 抓取策略

Hibernate使用 抓取策略 检索 相关对象在应用程序需要访问协会。 取策略可以被声明的O / R映射元数据,或 通过一个特定的HQL或来 标准 查询。

Hibernate3定义了以下抓取策略:

- » 连接抓取 :Hibernate检索 相关的实例或集合在同一 选择 ,使用一个 外 加入 。
- » 选择抓取 :第二 选择 用于检索相关的实体或 收集。 除非你显式禁用惰性抓取指定 懒= " false ",第二种选择只会 当你访问协会执行。
- » Subselect抓取 :第二 选择 用于检索相关的 为所有实体检索集合在之前的查询或获取。 除非你显式禁用惰性抓取指定 懒= " false ",第二种选择只会 当你访问协会执行。
- » 批量抓取 :一个优化策略 对于选择抓取。 Hibernate检索一组实体实例 或集合在一个单一的 选择 通过指定一个 主键或外键的列表。

Hibernate也区分:

- » 直接抓取 :一个协会, 收集或属性是拿来当老板立即 加载。
- » 延迟集合抓取 :一个集合 拿来当应用程序调用一个操作在那 收集。 这是默认为集合。
- » “额外的懒惰” 集合抓取 :集合的单个元素从数据库访问 作为需要。 Hibernate尽量不去拿整个收集到 除非绝对需要的内存。 适用于大 集合。
- » 代理抓取 :一个单值协会 当一个方法获取除了标识符调用getter是吗 在相关的对象。
- » “没有代理” 抓取 :一个单值 协会是拿来当实例变量被访问。 代理抓取相比,这种方法不懒; 协会是拿来即使只访问的标识符。 它 也更加透明,因为没有代理是可见的吗 应用程序。 这种方法需要buildtime字节码插装 并不是必需的。
- » 懒惰的属性获取 :一个属性或 单值的协会是拿来当实例变量 访问。 这种方法需要buildtime字节码插装 并不是必需的。

我们有两个正交的观念在这里: 当 是 该协会获取和 如何 它是获取。 这是 重要的是你不要混淆他们。 我们使用 取 到 调整性能。 我们可以使用 懒惰 定义一个合同 什么数据总是可用在任何一个特定的分离实例 类。

20 1 1. 处理懒惰协会

默认情况下,使用惰性抓取Hibernate3选择为集合 和懒惰的代理抓取为单值关联。 这些默认值 意义对于大多数协会在大多数应用程序。

如果你设置 hibernate默认批量抓取大小, Hibernate将使用批量抓取优化为惰性抓取。 这 优化也可以启用以更细的粒度。

请注意,访问延时关联以外的 上下文的一个开放的Hibernate会话将会导致异常。 对于 示例:

```
s = sessions.openSession();
```

```

Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

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

Integer accessLevel = (Integer) permissions.get("accounts"); // Error!

```

因为权限集合是没有初始化的时候 会话 是封闭的,而收集不能 加载它的状态。 Hibernate不支持懒惰 为分离对象初始化。 这可以是固定的 将代码从集合中读取到之前 事务被提交。

或者,您可以使用一个非延迟集合或协会,通过指定 懒= " false " 协会 映射。 然而,就算延迟初始化用于 几乎所有的集合和关联。 如果您定义太多非延迟 协会在你的对象模型,Hibernate将取整 数据库到内存中每个事务。

另一方面,你可以使用连接抓取,这是由非延迟 自然,而不是选择在一个特定的事务,取 我们将 现在解释如何定制抓取策略。 在 Hibernate3, 选择获取策略机制为单值是相同的 协会和集合。

20 1 2. 调谐获取策略

选择抓取(默认)是极容易受到N + 1 选择的问题,所以我们想启用连接抓取的 映射文档:

```

<set name="permissions"
    fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set>

```

```

<many-to-one name="mother" class="Cat" fetch="join"/>

```

这个 取 策略中定义的映射 文档影响:

- 检索通过 get() 或 load()
- 检索发生隐式地当一个协会 导航
- 标准 查询
- HQL查询如果 subselect 抓取是 使用

不管你使用的抓取策略,定义的 图是保证非延迟加载到内存。 这可能, 然而,导致一些即时的选择被用来执行 特定的HQL查询。

通常,映射文档不是用来定制抓取。 相反,我们保持默认的行为,并覆盖于一个特定的 事务,使用 离开加入取 在HQL。 这告诉 Hibernate来获取协会的 第一个选择热情,使用一个 外连接。 在 标准 查询API,您将使用 setFetchMode(FetchMode.JOIN) 。

如果你想改变所使用的抓取策略 get() 或 load() ,你可以使用 标准 查询。 例如:

```

User user = (User) session.createCriteria(User.class)
    .setFetchMode("permissions", FetchMode.JOIN)
    .add( Restrictions.idEq(userId) )
    .uniqueResult();

```

这是Hibernate的ORM解决方案相当于一些调用 “获取计划” 。

一个完全不同的方法 N + 1选择问题是 使用二级缓存。

20 1 3. 单端协会代理

为实现惰性抓取集合使用Hibernate的自己 实现持久的集合。 然而,一个不同的机制 懒惰的行为需要在单端关联。 目标 实体的协会必须代理。 Hibernate实现懒惰 初始化代理持久化对象使用运行时字节码 增强访问通过CGLIB包。

在创业初期,Hibernate3生成代理默认情况下所有 持久化类,并使用它们,使惰性抓取的 多对一 和 一对一 协会。

映射文件可以声明一个接口作为代理 接口的类,代理 属性。 默认情况下,Hibernate使用类的一个子类。 这个 代理类必须实现一个默认的构造函数至少包 可见性。 这个构造函数是建议所有持久 类。

有潜在的问题时要注意将这种方法扩展 对多态类。 例如:

```

<class name="Cat" proxy="Cat">
    .....
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>

```

首先,实例 猫 永远不会 浇灌到 domesticCat ,即使底层 实例的实例 domesticCat :

```

Cat cat = (Cat) session.load(Cat.class, id); // instantiate a proxy (does not hit the db)
if ( !cat.isDomesticCat() ) { // hit the db to initialize the proxy

```

```
DomesticCat dc = (DomesticCat) cat;    // Error!  
....  
}
```

其次,可以打破代理 = = :

```
Cat cat = (Cat) session.load(Cat.class, id);    // instantiate a Cat proxy  
DomesticCat dc =  
    (DomesticCat) session.load(DomesticCat.class, id); // acquire new DomesticCat proxy!  
System.out.println(cat==dc);                // false
```

然而,情况并不像看上去的那么糟糕。 甚至 虽然我们现在有两个引用不同的代理对象, 底层实例将仍然是相同的对象:

```
cat.setWeight(11.0); // hit the db to initialize the proxy  
System.out.println( dc.getWeight() ); // 11.0
```

第三,你不能使用CGLIB代理的一个 最后 类或一个类与任何 最后 方法。

最后,如果你的持久对象获得任何资源上 实例化(如在初始化或默认的构造函数),然后这些 资源也将获得代理。 代理类是一个实际的持久化类的子类。

这些问题都是由于在Java的基本限制 单继承模型。 为了避免这些问题你的持久 每个类必须实现一个接口,宣布其业务 方法。 你应该指定这些接口映射文件的地方 CatImpl 实现接口 猫 和 DomesticCatImpl 实现 接口 domesticCat。 例如:

```
<class name="CatImpl" proxy="Cat">  
.....  
  <subclass name="DomesticCatImpl" proxy="DomesticCat">  
    .....  
  </subclass>  
</class>
```

然后代理的实例 猫 和 domesticCat 可以返回 load() 或 迭代() 。

```
Cat cat = (Cat) session.load(CatImpl.class, catid);  
Iterator iter = session.createQuery("from CatImpl as cat where cat.name='fritz'").iterate();  
Cat fritz = (Cat) iter.next();
```

注意

列表() 通常不会返回 代理。

关系也懒洋洋地初始化。 这意味着你必须 声明任何类型的属性 猫 ,而不是 CatImpl 。

某些操作做 不 需要代理 初始化:

- » equals() :如果持久化类不 覆盖 equals()
- » hashCode() :如果持久化类确实 没有覆盖 hashCode()
- » 标识符的getter方法

Hibernate将检测持久化类,重写 equals() 或 hashCode() 。

通过选择 懒 = "没有代理" 而不是 默认 懒 = "代理" ,你可以避免的问题 铸字相关。 然而,buildtime字节码插装 是必需的,和所有操作将导致直接的代理吗 初始化。

20 1 4. 初始化集合和代理

一个 LazyInitializationException异常 将抛出的 Hibernate如果未初始化的集合或代理以外的访问 的范围 会话 ,即。 ,当实体 拥有集或有参考到代理的 分离的状态。

有时一个代理或集合之前需要被初始化 关闭 会话。 你可以初始化的力量 打电话 cat.getSex() 或 cat.getKittens()大小() ,例如。 然而,这 可以被混淆代码的读者,不方便吗 通用代码。

静态方法 hibernate初始化() 和 Hibernate.isInitialized() ,提供应用程序 与一个方便的工作方式与集合或迟缓初始化 代理。 hibernate初始化(猫) 将迫使 初始化一个代理, 猫 ,只要它的 会话 仍然是开放的。 hibernate初始化(cat.getKittens()) 有一个类似的效应对于收集的 小猫。

另一个选择是保持 会话 开放 直到所有必需的集合和代理服务器加载。 在一些 应用程序架构,特别是那些代码访问 数据使用 Hibernate,使用它的代码在不同 应用程序层或不同的物理过程,它可以是一个问题 确保 会话 打开一个集合 初始化。 有两种基本的方法来处理这个问题:

- » 在一个基于web的应用程序,一个servlet过滤器可以用来 关闭 会话 只有最后的一个用户 要求,一旦呈现的观点是完整的(公开会议针对 模式)。 当然,这 地方重要要求的正确性的异常处理 你的应用程序基础设施。 是极其重要的 会话 是关闭,交易结束了吗 然后返回给用户,甚至当一个例外发生在 渲染的视图。 看到Hibernate Wiki的例子 "开放会话在视图" 模式。
- » 在一个应用程序与一个单独的业务层,业务 逻辑必须 "准备" 所有的收藏品,web层需要之前 返回。 这意味着业务层应该加载所有的 数据和返回所有数据已经初始化的 演示/ web层,是需要一个特定的用例。 通常,应用程序调用 hibernate初始化() 对于每一个集合, 需要在web层(这叫必须发生在吗 会话关闭时)或检索收集急切地使用 Hibernate查询与 取 条款或 FetchMode.JOIN 在 标准。 这通常是容易如果你采用 命令 模式代替 会话Facade 。
- » 你也可以附加一个以前装载的对象到一个新的 会话 与 merge() 或 锁() 访问未初始化的集合之前 或其他代理。 Hibernate不,当然 应该 不是,做这个自动因为它 将引入临时事务语义。

有时你不想初始化一个大集合,但是 还需要一些关于它的信息,比如它的大小,例如,或一个 数据的子集。

您可以使用一组过滤器得到集合的大小 没有初始化:

```
( (Integer) s.createFilter( collection, "select count(*)" ).list().get(0) ).intValue()
```

这个 createFilter() 方法还用于 有效地获取一个集合的子集而不需要 初始化整个集合:

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResults(10).list();
```

20 1 5. 使用批量抓取

使用批量抓取,Hibernate可以加载几个未初始化 如果一个代理访问代理。 是一种优化的批量抓取 懒惰的选择抓取策略。 有两种方法你可以配置 批量抓取:在类级别和收集水平。

批量抓取类/实体是更容易理解。 考虑下面的例子:在运行时你有25 猫 实例装载在 会话, 和每个 猫 有一个参考的吗 所有者, 一个人。 这个人 类映射与代理, 懒= " true "。 如果你现在遍历所有猫 叫 getOwner() 在每个,Hibernate将,默认情况下, 执行25 选择 语句来检索代理 所有者。 你可以调整这一行为通过指定一个 批量大小 在映射的 人:

```
<class name="Person" batch-size="10">...</class>
```

Hibernate将只执行一个查询:模式是10, 10、 5。

你也可以使批量抓取的集合。 例如,如果 每个人 有一个懒惰的集合 猫 年代,和10人正在加载 会话, 遍历所有的人都将生成 10 选择 年代,一个用于每一个调用 getCats()。 如果你启用批量抓取的 猫 在映射的集合 人 ,Hibernate可以预取集合:

```
<class name="Person">
  <set name="cats" batch-size="3">
    ...
  </set>
</class>
```

与 批量大小 3,Hibernate将负载3, 3、 3、 1集合在4 选择 年代。再一次,值 的属性取决于预期数量的未初始化 集合在一个特定的 会话。

批量抓取的集合是特别有用如果你有一个 嵌套树的物品,即典型的材料清单模式。 然而,一个 嵌套 或 物化 路径 可能是一个更好的选择对于读为主的树木。

20 1 6. 使用subselect抓取

如果一个人懒集合或单值代理已经被取出, Hibernate会加载所有内容,重新运行原始查询在一个 subselect。 这个工作在同样的方式作为批量抓取但没有 零碎的加载。

20 1 7. 获取配置文件

另一种方式来影响加载相关抓取策略 对象是通过一种叫获取配置文件,这是一个命名的 配置相关 org.hibernate.sessionfactory 但是启用, 的名字,在 org.hibernate.会话。 一旦启用在 org.hibernate.会话, 获取配置文件将在影响的 org.hibernate.会话 直到它 显式禁用。

所以这是什么意思? 好让解释说,通过一个 的例子,展示了不同的可用的方法来配置 获取配置文件:

例20.1. 指定一个获取配置文件使用 @FetchProfile

```
@Entity
@FetchProfile(name = "customer-with-orders", fetchOverrides = {
    @FetchProfile.FetchOverride(entity = Customer.class, association = "orders", mode = FetchType.JOIN)
})
public class Customer {
    @Id
    @GeneratedValue
    private long id;

    private String name;

    private long customerNumber;

    @OneToMany
    private Set<Order> orders;

    // standard getter/setter
    ...
}
```


例20.2. 指定一个获取配置文件使用 <获取配置文件> 外 <类> 节点

```
<hibernate-mapping>
  <class name="Customer">
    ...
    <set name="orders" inverse="true">
      <key column="cust_id"/>
      <one-to-many class="Order"/>
    </set>
  </class>
  <class name="Order">
    ...
  </class>
  <fetch-profile name="customer-with-orders">
    <fetch entity="Customer" association="orders" style="join"/>
  </fetch-profile>
</hibernate-mapping>
```

例20.3. 指定一个获取配置文件使用 <获取配置文件> 里面 <类> 节点

```
<hibernate-mapping>
  <class name="Customer">
    ...
    <set name="orders" inverse="true">
      <key column="cust_id"/>
      <one-to-many class="Order"/>
    </set>
    <fetch-profile name="customer-with-orders">
      <fetch association="orders" style="join"/>
    </fetch-profile>
  </class>
  <class name="Order">
    ...
  </class>
</hibernate-mapping>
```

现在通常情况下当你得到一个引用一个特定的客户，客户的订单设置将懒惰我们将没有意义 从数据库加载这些订单。通常这是一件好事。现在让我们说，你有一个特定的用例，它更高效 加载客户和他们的订单在一起。当然是方法之一 使用“动态抓取”策略通过HQL或criteria查询。但另一个选择是使用一个获取配置文件来实现。以下代码将加载两个客户 和 他们 订单:

例20.4. 激活配置文件获取给定 会话

```
Session session = ...;
session.enableFetchProfile( "customer-with-orders" ); // name matches from mapping
Customer customer = (Customer) session.get( Customer.class, customerId );
```

注意

@FetchProfile 定义全局和 没关系,你把他们班。你可以把 @FetchProfile 要么到一个类或注释 包(包信息的java)。为了定义多个取 配置文件为同一类或包 @FetchProfiles 可以被使用。

目前只支持加入风格获取配置文件,但他们的计划是支持额外的风格。看到 [终极战士- 3414](#) 详情。

20 1 8. 使用延迟属性获取

Hibernate3支持惰性抓取的个人属性。这种优化技术也称为 取 组。请注意,这很可能是一个营销功能;优化行读取重要得多比优化的列 读取。然而,只有加载一些类的属性可能是有用的 在极端的情况下。例如,当遗留表有数百 列和数据模型不能得到改善。

让懒惰的属性加载,设置 懒惰 在你的特定属性的属性映射:

```
<class name="Document">
  <id name="id">
    <generator class="native"/>
  </id>
  <property name="name" not-null="true" length="50"/>
  <property name="summary" not-null="true" length="200" lazy="true"/>
```

```
<property name="text" not-null="true" length="2000" lazy="true"/>
</class>
```

懒惰的属性加载需要buildtime字节码插装。如果你的持久化类不增强,Hibernate将忽略懒惰 属性设置和返回直接抓取。

对字节码插装,使用下面的Ant任务:

```
<target name="instrument" depends="compile">
  <taskdef name="instrument" classname="org.hibernate.tool.instrument.InstrumentTask">
    <classpath path="{jar.path}"/>
    <classpath path="{classes.dir}"/>
    <classpath refxml:id="lib.class.path"/>
  </taskdef>

  <instrument verbose="true">
    <fileset dir="{testclasses.dir}/org/hibernate/auction/model">
      <include name="*.class"/>
    </fileset>
  </instrument>
</target>
```

一种不同的方法来避免不必要的列读取,至少 只读事务,是使用投影特征。HQL或 标准查询。 这避免了需要buildtime字节码处理 和当然是首选的解决方案。

你可以迫使通常使用预先抓取的属性 获取所有属性 在HQL。

20.2。 二级缓存

Hibernate 会话 是一个事务级缓存 持久存储的数据。 可以配置一个集群或jvm级别 (SessionFactory 根据程度的)缓存和 收集收集基础。 你甚至可以塞在集群缓存。 是 意识到缓存并没有意识到变化的持久性存储 另一个应用程序。 不过,它们可以被配置为定期到期 缓存的数据。

你可以选择告诉Hibernate,缓存 实现使用通过指定类的名称,它实现了 org.hibernate.cache.spi.CacheProvider 使用属性hibernate缓存提供者类。 Hibernate是捆绑 与一些内置集成与开源缓存 供应商列出的 表格20.1,“缓存提供者”。 你也可以实现自己的,把它插在如前所述。 注意, Hibernate 3.2之前版本使用EhCache作为默认的缓存 提供者。

表20.1。 缓存提供者

缓存	Provider类	类型	集群安全	查询缓存支持
ConcurrentHashMap(仅用于测试目的,在hibernate测试模块)	org.hibernate.testing.cache.CachingRegionFactory	内存		是的
EHCache	org.hibernate.cache.ehcache.EhCacheRegionFactory	内存、磁盘、事务性、集群	是的	是的
Infinispan	org.hibernate.cache.infinispan.InfinispanRegionFactory	集群(ip多播),事务	是的(复制或失效率)	是的(时钟同步点播。)

20 2 1。 缓存映射

正如我们在前几章所做的我们看着这两个 不同的possibilitites配置缓存。 第一次配置通过 注释,然后通过Hibernate映射文件。

默认情况下,实体不属于二级缓存和我们 建议你坚持这个设置。 然而,您可以覆盖这个 通过设置 共享缓存模式 元素在你 persistence . xml 文件或通过使用 javax.persistence.sharedCache.mode 财产在你 配置。 下面的值是可能的:

- ▶ 使选择性 (默认和推荐 值):实体不缓存,除非明确标记为 可缓存的。
- ▶ 禁用选择性 :实体缓存 除非显式地标记为不缓存。
- ▶ 所有 :所有实体总是缓存即使 标记为非缓存。
- ▶ 没有 :没有实体缓存即使标记 作为缓存。 这个选项可以意义二级伤残 缓存完全。

缓存的并发策略可以设置默认使用globaly 通过 hibernate缓存默认缓存并发策略 配置属性。 这个属性的值是:

- ▶ 只读

- » 读写
- » nonstrict-read-write
- » 事务性

注意

建议定义缓存并发策略/ 实体,而不是使用一个全局人。 使用 @org.hibernate.annotations.Cache 注释 那。

例20.5. 定义缓存并发策略通过 @Cache

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

Hibernate也让我们你缓存内容的收集或 标识符如果集合包含其他实体。 使用 @Cache 注释在收集 财产。




例20.6. 缓存集合使用注释

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

例20.7, “@Cache 注释与 属性” 显示 这个 @org.hibernate.annotations.Cache 注释与 它的属性。 它允许您定义缓存策略和地区 一个给定的二级缓存。

例20.7. @Cache 注释与 属性




```
@Cache(
    CacheConcurrencyStrategy usage();
    String region() default "";
    String include() default "all";
)
```

-  用法:给定的缓存并发策略(没有, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, 事务)
-  区域(可选):缓存区域(默认的fqcn 的类或角色的集合的名称)
-  包括 (可选):所有的包括所有 属性,非延迟只包括非延迟属性 (默认)。

现在让我们看看Hibernate映射文件。 那里的 <缓存> 元素的一个类或集合 映射是用于配置二级缓存。 看着 例20.8, “冬眠 <缓存> 映射 元素” 与此类似 annotations是显而易见的。

例20.8. Hibernate <缓存> 映射 元素

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"
    region="RegionName"
    include="all|non-lazy"
/>
```

-  使用 (需要)指定缓存 策略: 事务性, 读写, nonstrict-read-write 或 只读
-  地区 (可选:默认为类 或收集角色名):指定二级的名称 缓存区域
-  包括 (可选:默认 所有) 非延迟 :指定 ,属性的实体映射与 懒= " true " 不能缓存当 必须启用惰性抓取

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

现在让我们看看不同的用法 策略

20 2 2。策略:只读

如果你的应用程序需要读,但不能修改,实例持久化类,只读缓存可以被使用。这是最简单和最优执行策略。它甚至可安全使用在一个集群中。

20 2 3。策略:读/写

如果应用程序需要更新数据,一个读写缓存可能是适当的。这个缓存策略不应使用如果serializable事务隔离水平是必需的。如果缓存中使用JTA的环境,你必须指定属性 manager_lookup_class 和命名一个策略来获得JTA transactionManager。在其他环境中,你应该确保事务完成当会话关闭()或会话断开()被称为。如果你想使用这种策略在一个集群中,你应该确保底层的缓存实现支持锁定。内置缓存提供者不支持锁定。

20 2 4。策略:nonstrict读/写

如果应用程序只是偶尔需要更新数据(即如果这是绝对不可能的,两个事务将尝试更新同一项目同时),和严格的事务隔离是不必需的,一个 nonstrict-read-write 缓存可能适当的。如果缓存中使用JTA环境中,您必须指定 manager_lookup_class。在其他环境中,你应该确保在事务完成时会话关闭()或会话断开()被称为。

20 2 5。策略:事务

这个事务性缓存策略提供支持完全事务性缓存提供者如JBoss TreeCache。这样一个缓存只能用于一个JTA的环境,你必须指定 manager_lookup_class。

20 2 6。缓存提供者/并发策略的兼容性

重要

所有的缓存提供者支持所有缓存的并发性策略。

下面的表显示了哪些供应商是兼容的这并发策略。

表20.2。缓存并发策略支持

缓存	只读	nonstrict-read-write	读写	事务性
ConcurrentHashMap(不能用于生产使用)	是的	是的	是的	
EHCache	是的	是的	是的	是的
Infinispan	是的			是的

20.3。管理缓存

当你通过一个物体 save(), Update() 或 saveOrUpdate(), 和 每当你检索对象使用 load(), get(), 列表(), 迭代() 或 滚动(), 对象 添加到内部缓存的吗 会话。

当 flush() 随后称,国家的 该对象将被同步到数据库。如果你不希望 这种同步发生,或者如果你是处理一个巨大的数量的 对象和需要有效地管理内存, 驱逐() 方法可以用来删除对象和它的 从一级缓存集合。

例20.9。Explicitly驱逐一个缓存实例从第一级缓存 使用 会话驱逐()

```
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);
}
```

这个 会话 还提供了一个 包含() 方法来确定是否属于一个实例 到会话缓存。

驱逐所有对象的会话缓存,叫 会话清楚()

为第二级缓存,有方法上定义的 SessionFactory 将缓存的状态 实例,整个类,收集实例或整个集合 的角色。

例20.10。 二级缓存回收通过 SessionFactory.evict() 和 SessionFactory.evictCollection()

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class); //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections
```

这个 CacheMode 控制一个特定的会话 与第二级缓存:

- » CacheMode.NORMAL :将阅读物品和 编写项目的二级缓存
- » CacheMode.GET :将读取商品的 二级缓存。 不写入二级缓存除了当吗 更新数据
- » CacheMode.PUT :会写物品的 二级缓存。 不读从二级缓存
- » CacheMode.REFRESH :会写物品的 二级缓存。 不读从二级缓存。 旁路 的影响 hibernate缓存使用最小的把 迫使一个 刷新的二级缓存读取的所有商品 数据库

浏览内容的第二级或查询缓存区域,使用 这个 统计 API:

例20.11。 浏览二级缓存条目通过 统计 API

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

您需要启用统计数据,并可选地,迫使Hibernate 保持缓存条目更可读的格式:

例20.12。 使Hibernate统计

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

20.4。 查询缓存

查询结果集也可以被缓存。 这仅仅是有用的 经常运行的查询与相同的参数。

20.4.1。 启用查询缓存

缓存查询结果介绍了一些开销的角度 应用程序正常的事务处理。 例如,如果您的缓存 查询的结果对人Hibernate需要跟踪 当 这些结果应该无效,因为变化 违反的人。 加上这一事实,大多数 应用程序完全没有获得受益于缓存查询结果,领导 Hibernate禁用缓存默认的查询结果。 使用查询 缓存,您将首先需要启用查询缓存:

```
hibernate.cache.use_query_cache true
```

这个设置创建两个新的缓存区域:

- » org.hibernate.cache.internal.StandardQueryCache , 保持缓存的查询结果
- » org.hibernate.cache.spi.UpdateTimestampsCache , 持有时间戳的最新更新,可查询的表。 这些都是用于验证结果为 他们服务的 查询缓存。

重要

如果你配置你的潜在的缓存实现使用 过期或超时是非常重要的,缓存超时的 潜在的 UpdateTimestampsCache缓存区域被设置为一个 更高的价值比超时的任何查询缓存。 事实上,我们 推荐的UpdateTimestampsCache地区不被配置 在所有到期的时间。 注意,特别是,这一个LRU缓存失效 政策是不合适的。

如上所述,大多数查询不受益于缓存或 他们的研究结果。 所以在默认情况下,单个查询不缓存甚至 在启用查询缓存。 使结果缓存特定 查询、调用 org.hibernate.Query.setCacheable(真正的)。 这叫允许查询寻找现有缓存结果或添加它 结果当它执行

的缓存。

注意

查询缓存不缓存实际的实体的状态 在缓存中;它缓存只标识符值和结果的价值 类型。 对于这个原因,查询缓存应该总是被用于 结合二级缓存对于那些实体预计 作为缓存查询结果缓存 (就像收集 缓存)。

20 4 2. 查询缓存区域

如果你需要查询缓存过期的细粒度的控制 政策,您可以指定一个缓存区域命名为一个特定的查询 打电话 Query.setCacheRegion() 。

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

如果你想强迫查询缓存刷新它的一个地区 (无视任何缓存结果发现)可以使用 org.hibernate.Query.setCacheMode(CacheMode.REFRESH) 。 结合该地区定义为给定的查询, Hibernate将有选择地力结果缓存存在特定的 地区被刷新。 这是特别的情况下有用 底层的数据可能已经被更新通过一个单独的进程,是一个 更有效的替代大部分地区的驱逐通过 org.hibernate.SessionFactory.evictQueries() 。

20.5. 理解集合的性能

在前面的章节中我们已经介绍了集合和他们的 应用程序。 在本节中我们将探讨一些问题有关 在运行时集合。

20 5 1. 分类

Hibernate定义了三个基本类型的集合:

- 值的集合
- 一对多关联
- 多对多关联

这种分类区分各种表和外国 键关系但不告诉我们每件事我们需要知道 关于关系模型。 为了充分理解关系结构 和性能特征,我们也必须考虑到结构的 主键是用Hibernate来更新或删除收藏 行。 这表明以下分类:

- 索引集合
- 集
- 袋

所有索引集合(地图、列表和数组)有一个主 键组成的 <键> 和 <指数> 列。 在这种情况下,集合 更新是非常有效的。 主键可以有效 索引和一个特定的行可以有效地坐落在冬眠 尝试更新或删除它。

集有一个主键组成 <键> 和元素列。 这可以减少 有效的对一些类型的集合元素,尤其是复合 元素或大型文本或二进制字段,如数据库可能不可以 一个复杂的主键索引有效。 然而,对于一对多 或者多对多关联,特别是在案件的合成 标识符,它很可能会同样有效。 如果你想要 SchemaExport 实际创建主键的 <设置> ,你必须声明所有栏目 过的非null = " true " 。

< idbag > 映射定义一个代理键,所以他们是有效的更新。 事实上,他们是最好的案例。

袋是最坏的情况下,因为他们允许重复的元素值 和,因为他们没有索引列,没有主键可以被定义。 Hibernate已经没有办法区分重复的行。 Hibernate 解决这个问题,在一个单一的完全移除 删除 只要和再现集合 的变化。 这可能是低效的。

一对多关联,“主键”可能不是 物理数据库表的主键。 即使在这种情况下,上面的 分类仍然是有用的。 它反映出 Hibernate “定位” 个人的行集合。

20 5 2. 列表,地图,idbags和集是最有效的集合 更新

从上面的讨论,我们可以很明显的看出索引 集合和集合允许最有效的操作方面 添加、删除和更新的元素。

有,可以说,一个优势,索引集合 有集多对多关联或收藏的价值。 因为结构的一个 集,Hibernate不 更新 一行,当一个元素是“改变”。 改变 一个 集 总是工作通过 插入 和 删除 个人的行。 再一次,这 考虑并不适用于一对多关联。

进行观察后认为,数组不能懒惰,你可以断定 列表,地图和idbags是最有效的(非逆)集合 类型,设置不甘落后。 你可以期待集是最常见的一种集合在Hibernate应用程序。 这是因为 “设置” 的语义是最自然的关系模型。

然而,在精心设计的Hibernate域模型,大多数 集合是事实上一对多关联与 逆= " true " 。 对于这些协会、更新 由多对一关联的结束,所以考虑 的收集更新性能完全不适用。

20 5 3。 袋和列表是最有效的逆集合

有一个特定的情况下,然而,在这袋,也 列表,更高效的比集。 为一个集合 逆= " true " ,标准的双向 一对多关系的成语,例如,我们可以添加元素 袋或列表,而不需要初始化(取)袋子的元素。 这是,因为与一个吗 集,集合添加() 或 收集addAll() 必须始终返回 true—袋吗 或 列表 。 这可以使以下常见的代码太多 速度:

```
Parent p = (Parent) sess.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c); //no need to fetch the collection!
sess.flush();
```

20 5 4。 一次删除

一个一个删除集合元素有时会非常 效率低下。 Hibernate不知道做,在案件的一个 新空集合(如果你叫 列表清楚(), 例如)。 在这种情况下,Hibernate将发布一个 删除 。

假设您添加一个元素的集合大小二十 然后删除两个元素。 Hibernate将发行一 插入 语句和两个 删除 语句,除非集合是一个 袋。 这当然是 可取的。

然而,假设我们删除元素,留下两个,十八岁 然后添加你的新元素。 有两种可能的方法 进行

- » 删除18行一个接一个,然后插入三个 行
- » 删除整个集合在一个SQL 删除 并插入当前元素一个五 一个

Hibernate不能知道,第二个选择是可能更快。 它可能是不受欢迎的Hibernate,直观 这样的行为可能会混淆数据库触发器等。

幸运的是,您可以迫使这种行为(即第二 策略)在任何时候通过丢弃(即废弃)原始 收集和返回一个新实例化的集合 当前元素。

一枪删除不适用于集合映射 逆= " true " 。

20.6。 监控性能

优化并没有太大的使用没有监控和访问 性能数据。 Hibernate提供了全面的数据对其 内部操作。 统计在Hibernate可用每 SessionFactory 。

20 6 1。 监测SessionFactory

你可以访问 SessionFactory 指标在两个 的方式。 你的第一个选择是调用 SessionFactory.getStatistics() 和读或显示 这个统计 你自己。

Hibernate也可以使用JMX出版指标如果你启用 StatisticsService MBean。 你可以使一个单一的 MBean因你一切的 SessionFactory 或一个工厂。 看下面的代码为简约配置实例:

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

你可以激活和停用监视一个 SessionFactory :

- » 在配置时,设置 hibernate生成统计 到 假
- » 在运行时: sf.getStatistics().setStatisticsEnabled(真正的) 或 hibernateStatsBean.setStatisticsEnabled(真正的)

统计可以重置以编程方式使用的 clear() 法。 总结可以被发送到一个日志 (信息级)使用 logSummary() 法。

20 6 2. 指标

Hibernate提供了一个数量的指标,从基本信息 更多的专业信息,只是在某些情况下,相关 所有可用的计数器中描述 统计 接口 API,在三个类别:

- ▶ 度量相关的一般 会话 使用,如数量的开放课程,检索到的JDBC连接, 等。
- ▶ 度量相关的实体、集合的查询,和 缓存作为一个整体(即全球指标)。
- ▶ 详细的指标与一个特定的实体、收集、 查询或缓存区域。

例如,您可以检查缓存命中,小姐,把比率 实体,收集和查询,查询需要的平均时间。 要知道的毫秒数近似服从 java。 Hibernate 是绑在JVM精度和在某些平台上这个 可能只有精确到10秒。

简单的getter方法是用于访问全球指标(即不是 绑定到一个特定的实体、收集、缓存区域等)。 你可以 访问一个特定的实体的 指标,收集或缓存区域 通过它的名称,并通过其HQL或SQL查询表示。 请参考 统计 , EntityStatistics , CollectionStatistics , SecondLevelCacheStatistics ,和 QueryStatistics Javadoc API的更多信息。 这个 下面的代码是一个简单的例子:

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
    queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
    stats.getEntityStatistics( Cat.class.getName() );
long changes =
    entityStats.getInsertCount()
    + entityStats.getUpdateCount()
    + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times" );
```

你可以在所有实体、集合的查询和区域 缓存,通过检索名称列表的实体、集合 查询和区域缓存使用下面的方法: getQueries() , getEntityNames() , getCollectionRoleNames() ,和 getSecondLevelCacheRegionNames() 。

21章。 工具集导

表的内容

21.1. 自动模式生成

- 21 1 1. 定制模式
- 21 1 2. 运行该工具
- 21 1 3. 属性
- 21 1 4. 使用Ant
- 21 1 5. 增量模式更新
- 21 1 6. 使用Ant的增量模式更新
- 21 1 7. 模式验证
- 21 1 8. 使用Ant的模式验证

双向工程与Hibernate可以使用一组Eclipse插件, 命令行工具和Ant任务。

hibernate工具 目前包括插件的Eclipse IDE以及Ant任务为逆向工程现有的数据库:

- ▶ 映射编辑器: 一个编辑XML映射文件,Hibernate 支持自动完成和语法高亮显示。 它还支持语义 自动完成对类名和属性/字段名称,使其更加多才多艺的比普通的XML编辑器。
- ▶ 控制台: 控制台是一个新的视图在Eclipse中。 除了 一个树的概述你的控制台配置,你也提供一个交互式视图 你的持久化类和他们的关系。 控制台允许你 对数据库执行HQL查询和浏览结果直接在 eclipse。
- ▶ 开发向导: 几个向导提供了 Hibernate Eclipse工具。 您可以使用向导来快速生成Hibernate配置 (cfg xml)文件,或反向工程现有的数据库模式 到POJO源文件和Hibernate映射文件。 逆向工程向导 支持可定制的模板。
- ▶

请参考 hibernate工具 包文档 为更多的信息。

然而,Hibernate主要包是捆绑在一个集成的工具: SchemaExport Aka hbm2ddl 。 它甚至可以 从“内部”使用 Hibernate。

21.1. 自动模式生成

可以生成DDL从你由一个Hibernate映射文件实用程序。 生成的 模式包括引用完整性约束、主键和外键,因为 实体和收集表。 表和序列也创建映射 标识符生成器。

你 必须 指定SQL 方言 通过 hibernate方言 房地产当使用这个工具,因为DDL 是非常特定于供应商的。

首先,你必须定制你的映射文件来提高生成的模式。 下一节将介绍模式定制。

21.1.1. 定制模式

许多Hibernate映射元素定义可选属性命名 长度, 精密 和 规模。 你可以设置长度、精度 和规模的列的这个属性。

```
<property name="zip" length="5"/>
```

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

一些标签也接受 过的非null的 属性生成一个 非空 约束表的列上,和一个 独特的 属性生成 独特的 约束表的列上。

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

一个 独特的关键 属性可以用来组列 一个单一的、独特的关键约束。 目前的指定值 独特的关键 属性是 不 使用 命名约束在 生成的DDL。 它是只用于集团列 映射文件。

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

一个 指数 属性指定一个索引的名称 将创建使用映射的列或列。 多个列可以 分成相同的索引只需要通过指定相同的索引名称。

```
<property name="lastName" index="CustName"/>  
<property name="firstName" index="CustName"/>
```

一个 外键 属性可用于覆盖的名字 任何生成的外键约束。

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

许多映射元素也接受一个孩子 <列> 元素。 这是特别有用的多列映射类型:

```
<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>
```

这个 默认 属性允许您指定一个默认值 一个列。 你应该分配相同的值映射属性之前 保存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>
```

这个 sql类型 属性允许用户覆盖默认的 Hibernate映射类型的SQL数据类型。

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

这个 检查 属性允许您指定一个检查约束。

```
<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>
```

下面的表总结了这些可选的属性。

表21.1. 总结

属性	值	解释
长度	号码	柱长度
精密	号码	列小数精度
规模	号码	列十进制
过的非null的	真 假	指定列应该非空
独特的	真 假	指定列应该有一个独特的约束
指数	索引名称	指定的名称(多列)指数

独特的关键	独特的键名	指定一个多列的名称唯一约束
外键	外键名称	指定名称的外键约束生成的 对于一个协会,一个 <一对一> , <多对一的> , <键> , 或 <多对多> 映射元素。注意, 逆 = " true " 双方将不会被考虑 通过 SchemaExport 。
sql类型	SQL列类型	覆盖默认的列类型(属性 <列> 元素只有)
默认	SQL表达式	指定一个默认值列
检查	SQL表达式	创建一个SQL检查约束在两列或表

这个 <评论> 元素允许您指定的评论 生成的模式。

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

```
<property name="balance">
  <column name="bal">
    <comment>Balance in USD</comment>
  </column>
</property>
```

这个结果在一个 评论表 或 评论专栏 语句生成的 DDL支持。

21 1 2. 运行该工具

这个 SchemaExport 写一个DDL脚本工具到标准输出和/或 执行DDL语句。

下面的表显示了 SchemaExport 命令行选项

java - cp hibernate类路径 org.hibernate.tool.hbm2ddl.SchemaExport 选择映射文件

表21.2. SchemaExport 命令行选项

选项	描述
—安静	不输出脚本到stdout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
—= eg.MyNamingStrategy命名	选择一个 namingStrategy
—配置= 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=;	设置一个行将就木的分隔符为脚本

你甚至可以嵌入 SchemaExport 在你的应用程序:

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

21 1 3. 属性

数据库属性可以指定:

- » 作为系统属性与 - d <属性>
- » 在 hibernate属性
- » 在一个指定的属性文件 ——属性

所需的属性是:

表21.3. SchemaExport连接属性

属性名	描述
hibernate连接驱动程序类	JDBC驱动程序类
hibernate连接url	JDBC URL
hibernate连接用户名	数据库用户
hibernate连接密码	用户密码
hibernate方言	方言

21 1 4. 使用Ant

你可以叫 SchemaExport 从您的Ant构建脚本:

```
<target name="schemaexport">
  <taskdef name="schemaexport"
    classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
    classpathref="class.path"/>

  <schemaexport
    properties="hibernate.properties"
    quiet="no"
    text="no"
    drop="no"
    delimiter=";"
    output="schema-export.sql">
    <fileset dir="src">
      <include name="**/*.hbm.xml"/>
    </fileset>
  </schemaexport>
</target>
```

21 1 5. 增量模式更新

这个 SchemaUpdate 工具将会更新现有的模式与“增量”的变化。这个 SchemaUpdate 取决于JDBC元数据API,因此,将不使用所有JDBC驱动程序。

java - cp hibernate类路径 org.hibernate.tool.hbm2ddl.SchemaUpdate 选择映射文件

表21.4. SchemaUpdate 命令行选项

选项	描述
—安静	不输出脚本到stdout吗
—文本	不导出脚本到数据库吗
—= eg.MyNamingStrategy命名	选择一个 namingStrategy
—属性= hibernate属性	从文件读取数据库属性
—配置= hibernate cfg xml	指定一个 cfg xml 文件

你可以嵌入 SchemaUpdate 在你的应用程序:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

21 1 6. 使用Ant的增量模式更新

你可以叫 SchemaUpdate 从Ant脚本:

```
<target name="schemaupdate">
  <taskdef name="schemaupdate"
    classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
    classpathref="class.path"/>

  <schemaupdate
    properties="hibernate.properties"
    quiet="no">
    <fileset dir="src">
      <include name="**/*.hbm.xml"/>
    </fileset>
  </schemaupdate>
</target>
```

21 1 7. 模式验证

这个 SchemaValidator 工具将会验证现有的数据库模式“匹配”你的映射文件。这个 SchemaValidator 很大程度上取决于JDBC 元数据API,因此,不会处理所有JDBC驱动程序。这个工具是非常有用的用于测试。

java - cp hibernate类路径 org.hibernate.tool.hbm2ddl.SchemaValidator 选择映射文件

下面的表显示了 SchemaValidator 命令行选项:

表21.5. SchemaValidator 命令行选项

选项	描述
—= eg.MyNamingStrategy命名	选择一个 namingStrategy
—属性= hibernate属性	从文件读取数据库属性
—配置= hibernate cfg.xml	指定一个 cfg.xml 文件

你可以嵌入 SchemaValidator 在你的应用程序:

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

21 1 8. 使用Ant的模式验证

你可以叫 SchemaValidator 从Ant脚本:

```
<target name="schemavalidate">
  <taskdef name="schemavalidator"
    classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"
    classpathref="class.path"/>

  <schemavalidator
    properties="hibernate.properties">
    <fileset dir="src">
      <include name="**/*.hbm.xml"/>
    </fileset>
  </schemavalidator>
</target>
```

22章。 额外的模块

表的内容

22.1. Bean验证

- 22日1 1. 添加Bean验证
- 22日1 2. 配置
- 22日1 3. 抓违规
- 22日1 4. 数据库模式

22.2. Hibernate搜索

- 22日2 1. 描述
- 22日2 2. 整合与Hibernate注释

Hibernate核心也提供集成一些外部 模块/项目。 这包括Hibernate验证框架的参考 实现Bean验证(JSR 303)和Hibernate搜索。

22.1. Bean验证

Bean验证规范如何定义和声明的域模型 水平约束。 你可以,例如,表示一个属性应该 永远不会空,帐户余额应该严格积极等。 这些域模型约束的声明bean本身通过 注释它的属性。 Bean验证可以阅读和检查 对于约束违反。 验证机制可以执行 不同的层在你的应用程序,无需复制任何 这些规则(表示层、数据访问层)。 干燥后 原则,Bean验证和参考实现Hibernate 验证器被设计为目的。

之间的集成Hibernate和Bean验证工作在两个 的水平。 首先,它可以检查内存中实例的类 约束违反。 第二,它可以应用约束 Hibernate的元模型,并纳入生成的数据库 模式。

每个约束注释是关联到一个验证器 实现负责检查约束的实体 实例。 一个验证器也可以(可选地)应用约束的 Hibernate的元模型,允许Hibernate生成DDL,表达了 约束。 与适当的事件侦听器,您可以执行 检查操作在插入、更新和删除通过 Hibernate。

当检查实例在运行时,Hibernate Validator返回 信息在一组约束违反 ConstraintViolation 年代。其他信息, ConstraintViolation 包含一个错误描述 消息,可以嵌入参数值的包与注释 (如。 大小限制),和消息字符串,可以外部化到 ResourceBundle 。

22日1 1. 添加Bean验证

使Hibernate的Bean验证集成,只需添加一个 Bean验证提供者(最好是Hibernate验证4)在你的 类路径。

22日1 2. 配置

默认情况下,没有配置是必要的。

这个 默认 集团是在实体进行验证 插入和更新和数据库模型的基础上进行相应的更新 这个 默认 集团。

你可以定制Bean验证集成通过设置 验证模式。 使用 `javax.persistence.validation.mode` 属性并设置它 例如在你了 `persistence.xml` 文件或你的 `hibernate.cfg.xml` 文件。 几个选项 可能的:

- ▶ 汽车 (默认):使之间的集成 Bean验证和Hibernate(回调和ddl生成)只有 Bean验证是目前在类路径中。
- ▶ 没有 :禁用所有Bean之间的集成 验证和Hibernate
- ▶ 回调 :只有验证实体当他们 要么是插入、更新或者删除。 会抛出一个异常如果 没有Bean验证提供者是出现在类路径中。
- ▶ DDL :只有约束应用于数据库 模式生成Hibernate。 如果没有会抛出一个异常 Bean验证提供者是出现在类路径中。 这个值是 没有定义的Java持久性规范和特定于 Hibernate。

注意

您可以使用两 回调 和 DDL 在一起通过设置属性 回调,ddl

```
<persistence ...>
<persistence-unit ...>
...
<properties>
  <property name="javax.persistence.validation.mode"
            value="callback, ddl"/>
</properties>
</persistence-unit>
</persistence>
```

这相当于 汽车 除了,如果没有 Bean验证提供者在场,会抛出一个异常。

如果你想验证不同的团体在插入、更新 和删除,使用:

- ▶ `javax.persistence.validation.group.pre-persist` : 组织验证当一个实体即将被持久化(默认 默认)
- ▶ `javax.persistence.validation.group.pre-update` : 组织验证当一个实体即将被更新(默认 默认)
- ▶ `javax.persistence.validation.group.pre-remove` : 组织验证当一个实体即将被删除(默认 没有集团)
- ▶ `org.hibernate.validator.group.ddl` :组 考虑应用约束数据库模式(默认 到 默认)

每个属性接受的完全限定类名 组织验证由一个逗号分开(,)

例22.1. 使用自定义组进行验证

```
<persistence ...>
<persistence-unit ...>
...
<properties>
  <property name="javax.persistence.validation.group.pre-update"
            value="javax.validation.group.Default, com.acme.group.Strict"/>
  <property name="javax.persistence.validation.group.pre-remove"
            value="com.acme.group.OnDelete"/>
  <property name="org.hibernate.validator.group.ddl"
            value="com.acme.group.DDL"/>
</properties>
</persistence-unit>
</persistence>
```

注意

你可以设置这些属性 `hibernate.cfg.xml`, `hibernate`属性 或编程方式。

22日1 3. 抓违规

如果一个实体被发现是无效的,列表的约束 违规的传播 `ConstraintViolationException` 这暴露了 组 `ConstraintViolation` 年代。

这个异常是包裹在一个 `RollbackException` 当违反发生在 提交时间。 否则 `ConstraintViolationException` 返回(当调用示例 `flush()`)。 注意,一般来说,明显违反上级进行验证(例子在缝/ JSF 2通过JSF - Bean验证集成或 你的业务层通过显式地调用

Bean验证)。

应用程序代码将很少被寻找 ConstraintViolationException 提出通过Hibernate。 这个异常应该被视为致命和持久性上下文应该丢弃(EntityManager 或 会话)。

22日1 4。 数据库模式

Hibernate使用Bean验证约束生成一个精确的 数据库模式:

- » @NotNull 导致非Null列 (除非它冲突与组件或表继承)
- » @Size.max 导致 varchar(max) 定义字符串
- » 被, @Max 铅 到列检查(如 值< = max)
- » @Digits 领导的定义 精度和规模(你想知道哪个是哪个? 现在很容易 与 @Digits :))

这些约束可以被声明直接在实体 属性或间接通过使用约束组成。

更多信息查看Hibernate验证框架的参考文档在 <http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/>

22.2。 Hibernate搜索

22日2 1。 描述

全文搜索引擎喜欢 apache lucene 是一个非常强大的技术吗 把自由文本/高效的查询应用程序。 如果存在一些不匹配在处理一个 对象领域模型(保持索引更新,之间不匹配的索引结构和域 模型、 查询不匹配.....) Hibernate搜索索引你的域模型由于一些注释, 负责数据库/指数同步,让你回到常规管理的对象 自由文本查询。 Hibernate搜索使用 [apache lucene](#) 在后台。

22日2 2。 整合与Hibernate注释

Hibernate Search集成了Hibernate核心透明 如果Hibernate搜索类路径上存在罐子。 如果 你不希望自动注册Hibernate搜索事件 侦听器,您可以设置 hibernate.search.autoregister_listeners 为false。 这样一个需要很少见,不推荐。

检查Hibernate搜索参考文档(<http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/>)更多信息。

23章。 例如:父/子

表的内容

- [23.1。 一个注意集合](#)
- [23.2。 双向一对多](#)
- [23.3。 层叠生命周期](#)
- [23.4。 小瀑布和 未保存的价值](#)
- [23.5。 结论](#)

的第一件事情,新用户想做与Hibernate是模型一个父/子类型 关系。 有两种不同的方法来这。 最方便的 的方法,特别是对于新用户,是模型 父 和 孩子 作为实体类与一个 <一对多> 协会从 父 到 孩子 。 另一种方法是声明 孩子 作为一个 <复合元素> 。 默认的语义的一对多 协会在Hibernate不太接近通常语义的父/子关系比 一个复合元素的映射。 我们将解释如何使用 双向一对多 协会有下垂 建模一个父/子关系有效和优雅。

23.1。 一个注意集合

Hibernate集合被认为是一个逻辑部分拥有实体,不是的 包含实体。 注意,这是一个关键的区别,有以下结果:

- » 当你删除/添加一个对象从/到一个集合,版本号集合的所有者 是递增的。
- » 如果一个对象被从一个集合是一个值类型的实例(例如,一个复合 元素),该对象将不再是持久的,其状态将被完全移除 数据库。 同样地,添加一个值类型实例集合会导致其状态是 立即持久。
- » 相反,如果一个实体被删除从一个集合(一对多或多对多的 协会),它将被删除默认情况下。 这种行为是完全一致的;一个 改变另一个实体的内部状态不应导致关联实体消失。 同样地,添加一个实体集合不会引起,实体成为持久,通过 默认的。

添加一个实体集合,默认情况下,仅仅之间创建链接 这两个实体。 删除实体将删除链接。 这是适合所有类型的情况下。 然而,它是不合适的对于父/子关系。 在这种情况下,生活的 孩子被绑定到生命周期的父。

23.2. 双向一对多

假设我们从一个简单的 <一对多> 协会从 父 到 孩子。

```
<set name="children">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
```

如果我们执行下面的代码:

```
Parent p = ....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate将发布两个SQL语句:

- » 一个 插入 创建的记录 C
- » 一个 更新 创建链接 p 到 C

这不仅效率低下,而且还违反了任何 非空 约束在 父id 列。 你可以解决通过指定为空性约束违反 过的非null = " true " 在集合映射:

```
<set name="children">
  <key column="parent_id" not-null="true"/>
  <one-to-many class="Child"/>
</set>
```

然而,这是不推荐的解决方案。

这个行为的根本原因是,链接(外键 父id) 从 p 到 C 是不被认为是部分的状态吗 孩子 对象,因此不是创建的 插入 。 这个 解决方案是使链接的部分 孩子 映射。

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

你还需要添加 父 财产 孩子 类。

现在, 孩子 实体是管理国家的链接,我们告诉集合 不更新链接。 我们使用 逆 属性:

```
<set name="children" inverse="true">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
```

下面的代码将被用于添加一个新的 孩子 :

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

只有一个SQL 插入 现在将被发布。

你也可以创建一个 addChild() 方法 父 。

```
public void addChild(Child c) {
  c.setParent(this);
  children.add(c);
}
```

代码以添加一个 孩子 看起来像这样:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

23.3. 层叠生命周期

你可以解决挫折的显式的调用 save() 通过 使用小瀑布。

```
<set name="children" inverse="true" cascade="all">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
```

这简化了上面的代码:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

同样,我们不需要迭代的孩子在保存或删除 父。 下面的删除 p 和所有的孩子从数据库。

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

然而,下面的代码:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

不会删除 C 从数据库。 在这种情况下,它只会删除链接 p 和导致 非空 约束违反。 您需要显式地 删除() 这个 孩子。

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

在我们的例子中,一个 孩子 离不开它的父。 所以如果我们删除 一个 孩子 从集合中,我们希望它被删除。 要做到这一点,我们必须 使用 级联= "所有删除孤儿" 。

```
<set name="children" inverse="true" cascade="all-delete-orphan">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
```

即使集合映射指定 逆= " true ",小瀑布是 仍然对集合进行迭代处理元素。 如果你需要一个对象被保存,通过级联删除或更新,您必须将它添加到集合。 它是足够的,简单调用 setParent() 。

23.4. 小瀑布和 未保存的价值

假设我们加载了一个 父 在一个 会话,做了一些修改 在UI动作和想坚持这些变化在一个新的会话调用 Update()。 这个 父 将包含一组的儿童,而且由于级联更新可用, Hibernate需要知道哪些孩子是新实例化并表示现有的行 数据库。 我们还假设两个 父 和 孩子 产生 标识符属性的类型 长。 Hibernate将使用标识符和 版本/时间戳属性值来确定哪个孩子都是新的。(见 11.7 节, "自动状态检测")。 在Hibernate3,不再需要它来指定 一个 未保存的价值 明确。

下面的代码将会更新 父 和 孩子 和插入 newChild :

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

这可能是适合的情况下生成的标识符,但分配的标识符 和复合标识符? 这是更加困难的,因为Hibernate不能使用该标识符属性区分一个新实例化的对象,一个标识符分配的用户,和一个 对象加载在之前的会话。 在这种情况下,Hibernate将使用时间戳或版本 财产,或实际上会查询二级缓存,或者最糟糕的情况下,数据库,来看看 行存在。

23.5. 结论

我们刚刚覆盖的部分可能会有点混乱。 然而,在实践中,它的工作原理很好地。 大多数Hibernate应用程序使用父/子模式在许多地方。

我们提到的另一个在第一段。 上述问题存在的情况 <复合元素> 映射,它已经完全的语义父/子 关系。 不幸的是,有两个大的局限性与复合元素类:复合元素 不能自己的集合,他们不应该孩子的任何实体以外的其他独特的家长。

24章. 例如:Weblog应用程序

表的内容

24.1. 持久化类

24.2. Hibernate映射

24.3. Hibernate代码

24.1. 持久化类

这里的持久化类代表一个weblog和发表的文章 在一个博客。 他们是被建模为一个标准的父/子 关系,但是我们将使用一个命令包,而不是一组:

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

24.2. Hibernate映射

XML映射现在简单。 例如:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
```

```

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

  <class
    name="Blog"
    table="BLOGS">

    <id
      name="id"
      column="BLOG_ID">

      <generator class="native"/>

    </id>

    <property
      name="name"
      column="NAME"
      not-null="true"
      unique="true"/>

    <bag
      name="items"
      inverse="true"
      order-by="DATE_TIME"
      cascade="all">

      <key column="BLOG_ID"/>
      <one-to-many class="BlogItem"/>

    </bag>

  </class>

</hibernate-mapping>

```

```

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

<hibernate-mapping package="eg">

  <class
    name="BlogItem"
    table="BLOG_ITEMS"
    dynamic-update="true">

    <id
      name="id"
      column="BLOG_ITEM_ID">

      <generator class="native"/>

    </id>

    <property
      name="title"
      column="TITLE"
      not-null="true"/>

    <property
      name="text"
      column="TEXT"
      not-null="true"/>

    <property
      name="datetime"
      column="DATE_TIME"
      not-null="true"/>

    <many-to-one
      name="blog"
      column="BLOG_ID"
      not-null="true"/>

  </class>

</hibernate-mapping>

```

24.3. Hibernate代码

下面的类演示了一些事情 我们可以用这些类使用Hibernate:

```

package eg;

import java.util.ArrayList;

```

```

import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String text)
        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text)
        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);

```

```

        blog.getItems().add(item);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return item;
}

public void updateBlogItem(BlogItem item, String text)
    throws HibernateException {

    item.setText(text);

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.update(item);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
}

public void updateBlogItem(Long itemid, String text)
    throws HibernateException {

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
        item.setText(text);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
}

public List listAllBlogNamesAndItemCounts(int max)
    throws HibernateException {

    Session session = _sessions.openSession();
    Transaction tx = null;
    List result = null;
    try {
        tx = session.beginTransaction();
        Query q = session.createQuery(
            "select blog.id, blog.name, count(blogItem) " +
            "from Blog as blog " +
            "left outer join blog.items as blogItem " +
            "group by blog.name, blog.id " +
            "order by max(blogItem.datetime)"
        );
        q.setMaxResults(max);
        result = q.list();
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return result;
}

public Blog getBlogAndAllItems(Long blogid)
    throws HibernateException {

    Session session = _sessions.openSession();
    Transaction tx = null;
    Blog blog = null;
    try {
        tx = session.beginTransaction();
        Query q = session.createQuery(
            "from Blog as blog " +
            "left outer join fetch blog.items " +

```

```

        "where blog.id = :blogid"
    );
    q.setParameter("blogid", blogid);
    blog = (Blog) q.uniqueResult();
    tx.commit();
}
catch (HibernateException he) {
    if (tx!=null) tx.rollback();
    throw he;
}
finally {
    session.close();
}
return blog;
}

public List listBlogsAndRecentItems() throws HibernateException {

    Session session = _sessions.openSession();
    Transaction tx = null;
    List result = null;
    try {
        tx = session.beginTransaction();
        Query q = session.createQuery(
            "from Blog as blog " +
            "inner join blog.items as blogItem " +
            "where blogItem.datetime > :minDate"
        );

        Calendar cal = Calendar.getInstance();
        cal.roll(Calendar.MONTH, false);
        q.setCalendar("minDate", cal);

        result = q.list();
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return result;
}
}

```

第25章。 例如:各种映射

表的内容

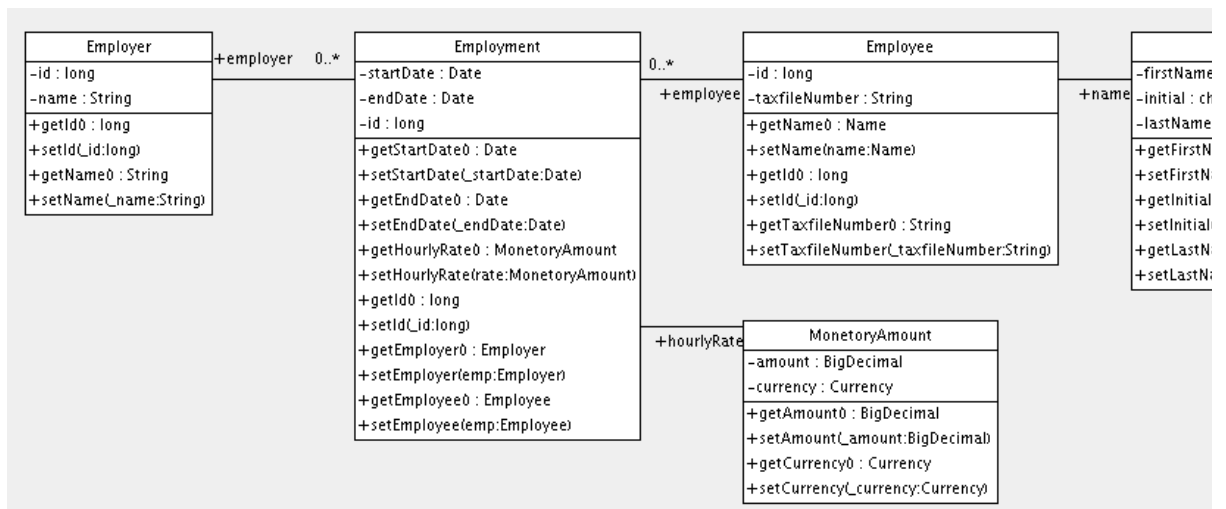
- 25.1. 雇主/雇员
- 25.2. 作者/工作
- 25.3. 客户/订单/产品
- 25.4. 杂项例子映射

- 25 4 1. “类型化” 一对一关联
- 25 4 2. 组合键的例子
- 25 4 3. 多对多与共享组合键属性
- 25 4 4. 基于内容的歧视
- 25 4 5. 协会在备用钥匙

本章探讨了一些更复杂的关联映射。

25.1. 雇主/雇员

下面的模型之间的关系 雇主 和 员工 使用一个实体类(就业) 代表协会。 你可以当可能有不止一个 工作期间,对于相同的两个政党。 组件是用于模型的货币 价值观和员工姓名。



这是一个可能的映射文档:

```

<hibernate-mapping>
  <class name="Employer" table="employers">
    <id name="id">
      <generator class="sequence">
        <param name="sequence">employer_id_seq</param>
      </generator>
    </id>
    <property name="name"/>
  </class>

  <class name="Employment" table="employment_periods">
    <id name="id">
      <generator class="sequence">
        <param name="sequence">employment_id_seq</param>
      </generator>
    </id>
    <property name="startDate" column="start_date"/>
    <property name="endDate" column="end_date"/>

    <component name="hourlyRate" class="MonetaryAmount">
      <property name="amount">
        <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
      </property>
      <property name="currency" length="12"/>
    </component>

    <many-to-one name="employer" column="employer_id" not-null="true"/>
    <many-to-one name="employee" column="employee_id" not-null="true"/>
  </class>

  <class name="Employee" table="employees">
    <id name="id">
      <generator class="sequence">
        <param name="sequence">employee_id_seq</param>
      </generator>
    </id>
    <property name="taxfileNumber"/>
    <component name="name" class="Name">
      <property name="firstName"/>
      <property name="initial"/>
      <property name="lastName"/>
    </component>
  </class>
</hibernate-mapping>
  
```

这是表模式所产生的 SchemaExport 。

```

create table employers (
  id BIGINT not null,
  name VARCHAR(255),
  primary key (id)
)

create table employment_periods (
  id BIGINT not null,
  hourly_rate NUMERIC(12, 2),
  currency VARCHAR(12),
  employee_id BIGINT not null,
  employer_id BIGINT not null,
  end_date TIMESTAMP,
  start_date TIMESTAMP,
  primary key (id)
)
  
```

```

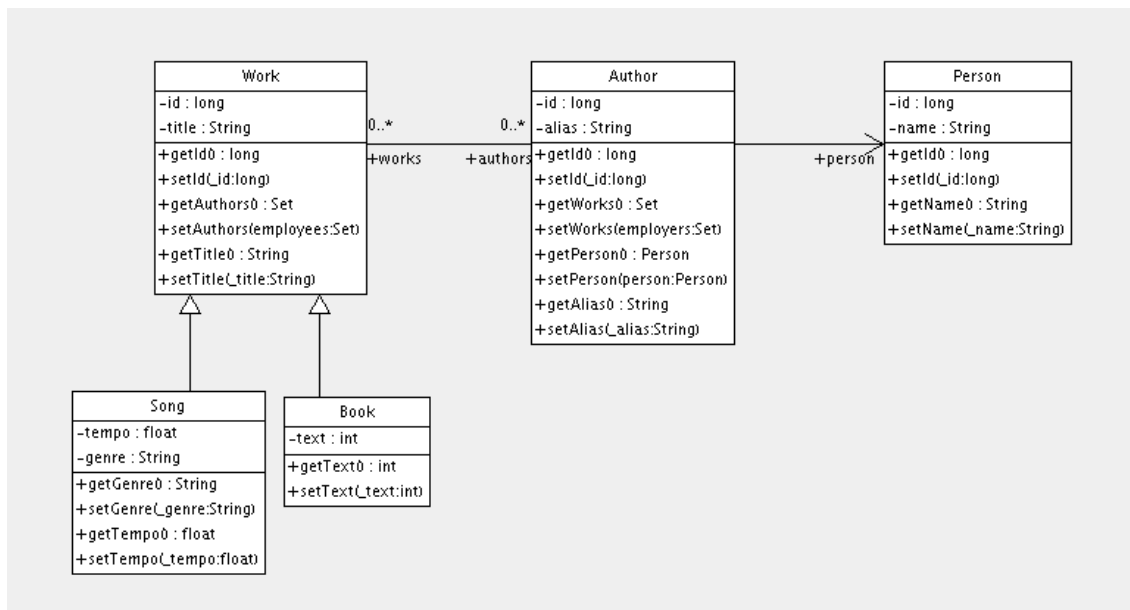
create table employees (
  id BIGINT not null,
  firstName VARCHAR(255),
  initial CHAR(1),
  lastName VARCHAR(255),
  taxfileNumber VARCHAR(255),
  primary key (id)
)

alter table employment_periods
  add constraint employment_periodsFK0 foreign key (employer_id) references employees
alter table employment_periods
  add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq

```

25.2. 作者/工作

考虑下面的模型之间的关系 工作, 作者 和 人。在这个例子中, 这个关系 之间 工作和 作者 被表示为一个多对多吗 协会的关系 作者 和 人 是表示为一对一关联。另一个可能性是 有 作者 扩展 人。



下面的映射文档正确地表示这些关系:

```

<hibernate-mapping>
  <class name="Work" table="works" discriminator-value="W">
    <id name="id" column="id">
      <generator class="native"/>
    </id>
    <discriminator column="type" type="character"/>
    <property name="title"/>
    <set name="authors" table="author_work">
      <key column name="work_id"/>
      <many-to-many class="Author" column name="author_id"/>
    </set>
    <subclass name="Book" discriminator-value="B">
      <property name="text"/>
    </subclass>
    <subclass name="Song" discriminator-value="S">
      <property name="tempo"/>
      <property name="genre"/>
    </subclass>
  </class>
  <class name="Author" table="authors">
    <id name="id" column="id">
      <!-- The Author must have the same identifier as the Person -->
      <generator class="assigned"/>
    </id>
    <property name="alias"/>

```

```

    <one-to-one name="person" constrained="true"/>

    <set name="works" table="author_work" inverse="true">
      <key column="author_id"/>
      <many-to-many class="Work" column="work_id"/>
    </set>

  </class>

  <class name="Person" table="persons">
    <id name="id" column="id">
      <generator class="native"/>
    </id>
    <property name="name"/>
  </class>

</hibernate-mapping>

```

有4个表在这个映射: 作品, 作者 和 人 保持工作, 作者 与人分别的数据。 作者工作 是一个协会 表链接作者的作品。 这是表模式, 所产生的 SchemaExport :

```

create table works (
  id BIGINT not null generated by default as identity,
  tempo FLOAT,
  genre VARCHAR(255),
  text INTEGER,
  title VARCHAR(255),
  type CHAR(1) not null,
  primary key (id)
)

create table author_work (
  author_id BIGINT not null,
  work_id BIGINT not null,
  primary key (work_id, author_id)
)

create table authors (
  id BIGINT not null generated by default as identity,
  alias VARCHAR(255),
  primary key (id)
)

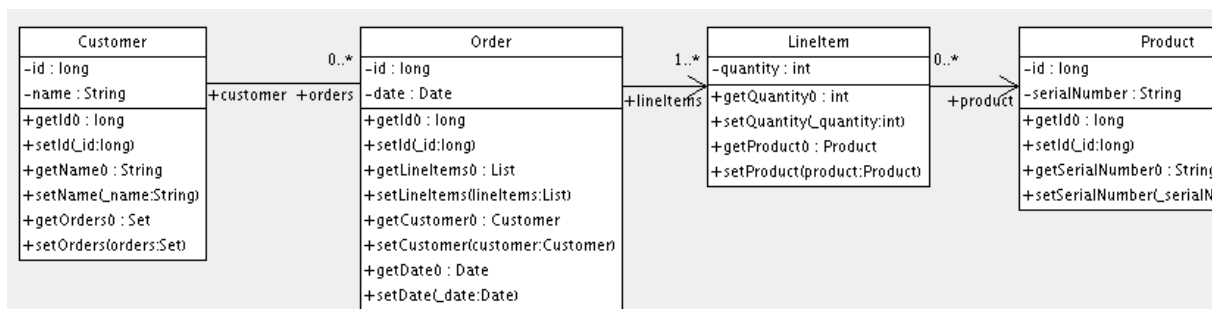
create table persons (
  id BIGINT not null generated by default as identity,
  name VARCHAR(255),
  primary key (id)
)

alter table authors
  add constraint authorsFK0 foreign key (id) references persons
alter table author_work
  add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
  add constraint author_workFK1 foreign key (work_id) references works

```

25.3. 客户/订单/产品

在这一节中, 我们考虑一个模型之间的关系 客户, 秩序, 行项目 和 产品。 有一个一对多的联系 客户 和 秩序, 但你怎么能代表 秩序 / LineItem / 产品 吗? 在这个例子中, LineItem 被映射为代表的多对多关联类的吗 联系 秩序 和 产品。 在 Hibernate 这称为复合元素。



映射文档将会看起来像这样:

```

<hibernate-mapping>

  <class name="Customer" table="customers">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="name"/>
    <set name="orders" inverse="true">

```



```

        <key column="customer_id"/>
        <one-to-many class="Order"/>
    </set>
</class>

<class name="Order" table="orders">
    <id name="id">
        <generator class="native"/>
    </id>
    <property name="date"/>
    <many-to-one name="customer" column="customer_id"/>
    <list name="lineItems" table="line_items">
        <key column="order_id"/>
        <list-index column="line_number"/>
        <composite-element class="LineItem">
            <property name="quantity"/>
            <many-to-one name="product" column="product_id"/>
        </composite-element>
    </list>
</class>

<class name="Product" table="products">
    <id name="id">
        <generator class="native"/>
    </id>
    <property name="serialNumber"/>
</class>

</hibernate-mapping>

```

客户, 订单, 行项目 和 产品 把握客户、订单、订单行项目和产品数据 分别。行项目 也作为表连接。协会 订单的产品。

```

create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders

```

25.4. 杂项例子映射

这些例子都可以从Hibernate测试套件。你会发现许多其他有用的例子映射通过搜索的有吗 测试 文件夹的Hibernate分布。

25 4 1. “类型化” 一对一关联

```

<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
        cascade="all">
        <formula>name</formula>
        <formula>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
        cascade="all">
        <formula>name</formula>
        <formula>'MAILING'</formula>
    </one-to-one>
</class>

```

```

<class name="Address" batch-size="2"
  check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
  <composite-id>
    <key-many-to-one name="person"
      column="personName"/>
    <key-property name="type"
      column="addressType"/>
  </composite-id>
  <property name="street" type="text"/>
  <property name="state"/>
  <property name="zip"/>
</class>

```

25 4 2. 组合键的例子

```

<class name="Customer">
  <id name="customerId"
    length="10">
    <generator class="assigned"/>
  </id>
  <property name="name" not-null="true" length="100"/>
  <property name="address" not-null="true" length="200"/>
  <list name="orders"
    inverse="true"
    cascade="save-update">
    <key column="customerId"/>
    <index column="orderNumber"/>
    <one-to-many class="Order"/>
  </list>
</class>
<class name="Order" table="CustomerOrder" lazy="true">
  <synchronize table="LineItem"/>
  <synchronize table="Product"/>
  <composite-id name="id"
    class="Order$Id">
    <key-property name="customerId" length="10"/>
    <key-property name="orderNumber"/>
  </composite-id>
  <property name="orderDate"
    type="calendar_date"
    not-null="true"/>
  <property name="total">
    <formula>
      ( select sum(li.quantity*p.price)
        from LineItem li, Product p
        where li.productId = p.productId
          and li.customerId = customerId
          and li.orderNumber = orderNumber )
    </formula>
  </property>
  <many-to-one name="customer"
    column="customerId"
    insert="false"
    update="false"
    not-null="true"/>
  <bag name="lineItems"
    fetch="join"
    inverse="true"
    cascade="save-update">
    <key>
      <column name="customerId"/>
      <column name="orderNumber"/>
    </key>
    <one-to-many class="LineItem"/>
  </bag>
</class>
<class name="LineItem">
  <composite-id name="id"
    class="LineItem$Id">
    <key-property name="customerId" length="10"/>
    <key-property name="orderNumber"/>
    <key-property name="productId" length="10"/>
  </composite-id>
  <property name="quantity"/>

```

```

<many-to-one name="order"
  insert="false"
  update="false"
  not-null="true">
  <column name="customerId"/>
  <column name="orderNumber"/>
</many-to-one>

<many-to-one name="product"
  insert="false"
  update="false"
  not-null="true"
  column="productId"/>

</class>

<class name="Product">
  <synchronize table="LineItem"/>

  <id name="productId"
    length="10">
    <generator class="assigned"/>
  </id>

  <property name="description"
    not-null="true"
    length="200"/>
  <property name="price" length="3"/>
  <property name="numberAvailable"/>

  <property name="numberOrdered">
    <formula>
      ( select sum(li.quantity)
        from LineItem li
        where li.productId = productId )
    </formula>
  </property>

</class>

```

25 4 3. 多对多与共享组合键属性

```

<class name="User" table="User">
  <composite-id>
    <key-property name="name"/>
    <key-property name="org"/>
  </composite-id>
  <set name="groups" table="UserGroup">
    <key>
      <column name="userName"/>
      <column name="org"/>
    </key>
    <many-to-many class="Group">
      <column name="groupName"/>
      <formula>org</formula>
    </many-to-many>
  </set>
</class>

<class name="Group" table="Group">
  <composite-id>
    <key-property name="name"/>
    <key-property name="org"/>
  </composite-id>
  <property name="description"/>
  <set name="users" table="UserGroup" inverse="true">
    <key>
      <column name="groupName"/>
      <column name="org"/>
    </key>
    <many-to-many class="User">
      <column name="userName"/>
      <formula>org</formula>
    </many-to-many>
  </set>
</class>

```

25 4 4. 基于内容的歧视

```

<class name="Person"
  discriminator-value="P">

  <id name="id"
    column="person_id"
    unsaved-value="0">

```

```

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

  <discriminator
    type="character">
    <formula>
      case
        when title is not null then 'E'
        when salesperson is not null then 'C'
        else 'P'
      end
    </formula>
  </discriminator>

  <property name="name"
    not-null="true"
    length="80"/>

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

  <component name="address">
    <property name="address"/>
    <property name="zip"/>
    <property name="country"/>
  </component>

  <subclass name="Employee"
    discriminator-value="E">
    <property name="title"
      length="20"/>
    <property name="salary"/>
    <many-to-one name="manager"/>
  </subclass>

  <subclass name="Customer"
    discriminator-value="C">
    <property name="comments"/>
    <many-to-one name="salesperson"/>
  </subclass>

</class>

```

25 4 5. 协会在备用钥匙

```

<class name="Person">
  <id name="id">
    <generator class="hilo"/>
  </id>

  <property name="name" length="100"/>

  <one-to-one name="address"
    property-ref="person"
    cascade="all"
    fetch="join"/>

  <set name="accounts"
    inverse="true">
    <key column="userId"
      property-ref="userId"/>
    <one-to-many class="Account"/>
  </set>

  <property name="userId" length="8"/>
</class>

<class name="Address">
  <id name="id">
    <generator class="hilo"/>
  </id>

  <property name="address" length="300"/>
  <property name="zip" length="5"/>
  <property name="country" length="25"/>
  <many-to-one name="person" unique="true" not-null="true"/>
</class>

<class name="Account">
  <id name="accountid" length="32">
    <generator class="uuid"/>
  </id>

  <many-to-one name="user"

```

```
column="userId"
property-ref="userId"/>

<property name="type" not-null="true"/>

</class>
```

26章。最佳实践

写细粒度的类和它们映射使用 <组件> :

使用一个 地址 类来封装 街, 郊区, 状态, 邮编。这有助于促进代码重用和简化重构。

声明标识符属性在持久化类:

Hibernate使标识符属性可选。有一系列的原因 你应该使用它们。我们建议标识符是“合成”,即生成 没有商业意义。

识别自然键:

识别自然键所有实体,并将它们映射使用 <自然id >。实现 equals() 和 hashCode() 比较属性构成的自然键。

每个类的地方在它自己的文件映射:

不要使用一个单一的整体映射文档。地图 com如foo 在 文件 com/eg/Foo.hbm.xml。这是有道理的,特别是在 一个团队环境中。

负载映射为资源:

部署映射随着类他们地图。

考虑外化查询字符串:

这是建议如果你的查询调用非ansi标准SQL函数。外部化字符串映射文件的查询将使应用程序更多 便携式。

使用绑定变量。

在JDBC,总是取代不恒定值由“?”。不要使用字符串操作 绑定一个不恒定值在一个查询。你也应该考虑使用命名参数查询。

不会管理自己的JDBC连接:

Hibernate允许应用程序管理JDBC连接,但是他的方法应该考虑 一个。如果你不能使用内置的联系供应商,考虑提供你自己实现的 org.hibernate.connection.ConnectionProvider。

考虑使用一个自定义的类型:

假设您有一个Java类型从一个库,需要坚持但不 提供访问器需要将其映射为一个组件。你应该考虑实现 org.hibernate.UserType。这种方法使应用程序 代码实现转换/从一个Hibernate类型。

使用手工编写JDBC在瓶颈:

在系统的性能关键的领域,一些种类的操作可能会从中受益 直接JDBC。不要假设,然而,JDBC必然是更快。请等待,直到你知道 一些是一个瓶颈。如果你需要使用直接JDBC,你可以打开一个Hibernate 会话,然后用你的JDBC操作作为一个 org.hibernate.jdbc工作 对象和使用JDBC连接。这 你仍然可以使用相同的事务策略和底层连接提供商。

理解 会话 冲洗:

有时会话同步它的持久状态与数据库。性能将 如果这个过程发生影响太经常。你有时可以减少不必要的冲洗的 禁用自动冲洗,甚至通过改变订单的查询和其他操作在一个 特定的事务。

在一个三层结构,考虑使用分离对象:

当使用一个servlet /会话bean的体系结构,您可以通过持久化对象载入 会话bean与servlet / JSP层。使用一个新会话,每个请求的服务。使用 会话合并() 或 Session.saveOrUpdate() 到 与数据库同步对象。

在一个两层结构,考虑使用长期持久性上下文:

数据库事务必须尽可能短,最佳的可伸缩性。然而,它通常是 有必要实现长时间运行 应用程序事务,一个单一的 工作单元的角度,一个用户。一个应用程序事务可能跨几个 客户端请求/响应周期。通常使用分离对象来实现应用程序 事务。一个合适的选择在一个两层结构,是维护 一个开放的持久性接触整个生命周期会议应用程序的事务。然后 简单的JDBC连接断开,在每年年底的请求和重新连接 后续请求的开始。从来没有共享一个会话跨多个应用程序 交易或你将使用陈旧的数据。

不要把异常作为可收回:

这更多的是一种必要的实践比一个“最佳”实践。当异常发生时,回滚 这个 事务 并关闭 会话。如果你不这样做,Hibernate 不能保证内存状态准确地代表了持久状态。例如,不要使用 会话负荷() 可以确定一个实例与给定的标识符存在于数据库;使用 会话获得() 或查询相反。

喜欢懒加载的关联:

少使用预先抓取。使用代理和懒惰的集合对于大多数协会类 不可能完全关押在二级缓存。对于关联到缓存的类,哪里有一个非常高的概率缓存命中的显式禁用使用预先抓取吗 懒 = " false "。当连接抓取是适合一个特定的使用 情况下,使用 一个查询用 离开加入取。

使用 公开会议针对 模式,或有纪律 装配阶段 为了避免问题,unfetched数据:

Hibernate使得开发人员编写单调乏味 数据传输对象 (DTO)。在传统的EJB架构,提供双重目的:首先dto,他们解决问题,实体bean不是序列化的;第二,他们隐式定义一个装配阶段 所有数据视图使用的抓取和编组到dto返回之前控制 到表示层。Hibernate消除第一目的。除非你准备举行 持久化上下文(会话)开放在视图呈现过程,你仍然需要 一个装配阶段。觉得你的业务方法具有严格的合同与演示 层对数据可用在分离的对象。这不是一个限制 冬眠的。这是一个基本要求的 安全事务数据访问。

考虑你的业务逻辑抽象从Hibernate:

隐藏Hibernate数据访问代码后面一个接口。结合 `刀` 和 `线程本地会话` 模式。你甚至可以有一些类的持久化 `handcoded JDBC` 关联到Hibernate通过 `userType`。这个建议是,然而,用于“充分大的”应用程序。它是不适合的 `应用程序 5` 表。

不要使用奇异的关联映射:

实际测试用例为真正的多对多关联是罕见的。大部分的时间你需要 额外的信息存储在“链接表”。在这种情况下,它是更好的来 使用两个一对多关联到一个中间环节类。事实上,是一对多和多对一大部分关联。因为这个原因,你应该谨慎行事当使用任何 其他协会的风格。

喜欢双向关联:

单向关联更难以查询。在一个大型的应用程序,几乎 所有的团体必须在两个方向上在查询通航。

第27章。数据库的可移植性的考虑

表的内容

- [27.1. 可移植性基本](#)
- [27.2. 方言](#)
- [27.3. 方言决议](#)
- [27.4. 标识符生成](#)
- [27.5. 数据库函数](#)
- [27.6. 类型映射](#)

27.1. 可移植性基本

的卖点之一Hibernate(和真正的对象/关系映射作为一个整体) 数据库的可移植性的概念。这可能意味着一个内部用户迁移从一个 数据库供应商到另一个,或者也可以是一个可部署的应用程序框架或消费 Hibernate同时目标多个数据库产品的用户。不管 确切的场景中,基本的想法是,你想要冬眠来帮助你在任何号码 数据库没有修改你的代码,和理想情况下没有任何更改映射元数据。

27.2. 方言

第一行的可移植性Hibernate是方言,这是一个专业化的 `org.hibernate` 方言方言 合同。一种方言封装了所有 的差异必须联系一家Hibernate数据库来完成一些 任务就像是做序列值或构建选择查询。Hibernate包范围广泛 对于许多的方言最流行的数据库。如果你发现你的特定的数据库 不是其中,它并不特别难写你自己的。

27.3. 方言决议

最初,Hibernate将总是要求用户指定哪些方言使用。对于 用户希望同时瞄准多个数据库与他们建立,是有问题的。通常这需要他们的用户配置Hibernate方言或定义自己的方法 设置这个值。

从版本3.2,Hibernate自动检测的概念引入的方言 使用基于 `java.sql` 该方法 取得 `java.sql` 连接到数据库。这是更好的,期待这个决议是限于数据库Hibernate提前了解,没有办法 可配置或 `overrideable`。

从版本3.3,Hibernate有票价更强大的方法来自动确定 应该使用的方言,依靠一系列的代表,它实现了吗 `org.hibernate.dialect.resolver.DialectResolver` 这只定义了一个 单一的方法:

```
public Dialect resolveDialect(DatabaseMetaData metaData) throws JDBCConnectionException
```

这里的基本合同,如果这个解析器”理解“给定的数据库元数据然后 它返回相应的方言,如果不是它返回null和流程继续接下来的 解析器。签名还标识 `org.hibernate.exception.JDBCConnectionException` 作为可能被抛出。一个 `JDBCConnectionException` 这里是解释为暗示一个“非瞬态”(又名不可恢复的)连接问题,用于指示立即停止解析 尝试。所有其他异常导致警告和继续到下一个解析器。

关于这些解析器的冷却部分是,用户也可以注册自己的自定义解析器 它将被处理之前的内置冬眠的。这可能是有用的在一个数量的 不同的情况:它允许容易集成的自动识别方言以外 附带Hibernate本身,它允许您指定使用一个自定义当一个特定的方言 数据库是公认的;等等。注册一个或多个解析器,只需指定它们(隔 逗号、制表符和空格)使用 `hibernate`。方言解析器的配置设置(参见 方言解析器 常数在 `org.hibernate.cfg` 环境)。

27.4. 标识符生成

当考虑可移植性数据库之间,另一个重要的决定就是选择 标识符生成您想要使用战略。最初Hibernate提供了 原生 发电机为这个目的,它是为了之间进行选择 一个 序列, 身份,或 表 策略取决于底层数据库的能力。然而,一个阴影的含义 这种方法是当 `targetting` 一些数据库支持 身份 生成和一些不。身份 生成依赖SQL 定义一个身份(或自动递增)列管理标识符值;它是什么 称为后插入生成策略because插入必须实际发生之前 知道该标识符值。因为Hibernate依赖这个标识符值来唯一地参考 实体 在一个持久化上下文必须然后发出插入 当用户请求立即entity关联的会话(如通过 `save()` 如。)无论当前的事务语义。

注意

Hibernate是一次的含义发生了微妙的变化,这是更好的理解 插入被延迟的情况下,是可行的。

潜在的问题是,实际的semantics应用程序自身的变化在这些情况下。

从版本3.2.3,Hibernate附带一组 **增强** 标识符生成器指向 可在一个相当不同的方式。

注意

有专门2捆绑 增强 发电机:

- » org.hibernate.id.enhanced.SequenceStyleGenerator
- » org.hibernate.id.enhanced.TableGenerator

这些发电机背后的想法是港口实际语义标识符的值

一代不同的数据库。

例如,

org.hibernate.id.enhanced.SequenceStyleGenerator 模仿行为的一个序列在数据库不支持序列通过使用一个表。

27.5. 数据库函数

警告

这是一个地区在Hibernate需要改进的。 可移植性问题而言, 这个函数处理目前工作很好从HQL;但是,它是相当缺乏 在所有其他方面。

SQL函数可以引用用户在很多方面。 然而,并非所有的数据库 支持相同的函数集。 Hibernate,提供了一种方法来映射一个 逻辑 函数名来代表,知道如何呈现 那个特定的功能,甚至使用一个完全不同的物理函数调用。

重要

从技术上讲,这是处理函数注册通过

org.hibernate.dialect.function.SQLFunctionRegistry 类 这是为了允许用户提供自定义的函数定义没有 必须提供一个自定义的方言。 这个特定的行为是没有完全完成 目前。 它是一种可以以编程的方式实现的,这样用户注册功能 与 org.hibernate.cfg配置 和那些功能 将被表彰HQL。

27.6. 类型映射

这部分的完成时间表在稍后的日期...

引用

[PoEAA] 企业应用架构模式。 0-321-12742-0 - -。 马丁 福勒。 版权©2003培生教育公司. . 出版社的出版公司。

[JPwH] Java持久性与Hibernate。 第二版的Hibernate在行动。 1-932394-88-5 - -。 <http://www.manning.com/bauer2>。 基督教 鲍尔 和 加文 王。 版权©2007曼宁出版有限公司. . 曼宁出版有限公司. .