# Hibernate 用户手册

## Hibernate - 纯 java 的关系型持久层框

Hibernate 团队

JBoss 可视化设计团队

5.0.0.Final



2015-08-20

# 目录

# 序言

面向对象的开发在处理关系型 数据库中的数据时是非常麻烦与消耗资源的。开发成本非常高的原因在于：关系型数据库中的数据与程序对象之间不匹配（译者注：数据库中的表如何变成程序的对象）。Hibernate 是 JAVA 开发环境下对象-关系映射（Object/Relational Mapping 及 ORM）的解决方案。ORM 是指一种映射技巧，ORM 适用于对象模型与关系模型之间的匹配。参考 Wikipedia（维基）的高级讨论组与 Martin Fowler 的文章 OrmHate，其中都提到了不匹配带来的问题。

虽然您可能认为已经有了强大的 SQL，hibernate 不是必需的，但是在做出判断前请了解一下 hibernate。同时，一些基本概念的理解也有助于你对 hibernate 更快更全面的了解。这些基本概念中，数据建模原则是非常重要的。下面两个文章有助于您了解数据建模原则。

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

了解一些基本的事务处理、设计模式比如："Unit of Work（工作单元）"[PoEAA]或者"ApplicationTransaction（应用程序事务）"是非常有益的。这些我们将在本文档中讨论研究，但是预先了解一些会带来很大的帮助。

Hibernate 不只关心 java 类到数据库表的映射（java 类型到 sql 类型的匹配），还提供了数据查询与检索工具。应用 hibernate 可以替换你手工处理 SQL 与 JDBC 的代码，它可以显著的减少开发时间。Hibernate 的设计目标是减少工程中９５％的 SQL 与 JDBC 操作。当然还有其它的数据持久化解决方案。但是 Hibernate 与其它方案不同在于它不阻断您使用强大的 SQL 与 JDBC，它只是希望您将更多的精力用于关系设计与业务逻辑上。

Hibernate 可能不是最好的 data-centric applications（以数据为中心应用程序）的解决方案，它只是用存储过程来实现数据库业务逻辑。但是它是一个 java 中间层，用于提供面向对象的建模与业务逻辑。Hibernate 可以帮助您去除与封装数据库提供商特定的 Sql 代码，帮助您完成将数据结果集从表格形式转换成对象的形式。

参考以下内容，得到相关信息：http://hibernate.org/orm/contribute/

提示

如果你是刚刚开始使用 Hibernate, 您可以从开始"**Hibernate Getting Started Guide（Hibernate 入门指南）**"开始文档页，它包含快速教程以及很多介绍。还有一系列的专题讨论，适应不同层次的读者。

# 第 1 章. Architecture（体系架构）

**目录**

## 1.1. 概述

Hibernate 作为一个 ORM 解决方案，实际上位于 java 应用与关系数据库之间。可以参考上图所示。JAVA 应用使用 Hibernate API 加载、存储、查询等操作，处理数据库中的数据。这里先简单介绍一下基本的 Hibernate API，稍后我们再讨论它的细节。

SessionFactory（会话工厂）(org.hibernate.SessionFactory)

它是一个线程安全的，immutable（终态的），它是一个代理，表示应用程序域模型到数据库的映射，作为建立 org.hibernate.Session（会话实例）的工厂。SessionFactory 的建立代价很大（译者注：内存、cpu 占用很大），所以一个应用只能有一个 SessionFactory。SessionFactory 维护 Hibernate 的所有 Session（会话），二级缓冲，连接池，事务等等。

Session（会话）(org.hibernate.Session)

Session（会话）是一个单线程，短生命周期的对象，是按"Unit of Work（工作单元）"[PoEAA]模式的概念构建的。

封装了 JBDC 的连接对象 java.sql.Connection。作为一个 org.hibernate.Transaction（事务）实例的工厂，维护应用程序 domain model（数据模型、域模型）中通用（"repeatable read 可重复读取"）的持久化内容（一级缓冲）。

Transaction（事务）(org.hibernate.Transaction)

它是单线程，短生命周期的对象，它用于界定具体物理事务的范围。它作为一个抽象 API 接口用于隔离各种应用与底层的事务系统（JDBC, JTA, CORBA……）。

## 1.2. Contextual sessions（session 上下文）

大多数应用程序使用 Hibernate 时需要某种形式的 session“上下文”，一个特定的 session(会话)影响整体特定的上下文范围。然而，跨应用程序的组建上下文是很难的；目前的观点是：不同的上下文定义了不同的范围。在 Hibernate3 以前，应用程序使用 session 有两种方式：一种是利用 ThreadLocal（本地线程）加上辅助类如：HibernateUtil，建立 session(会话)；另一种是使用第三方框架，如：Spring、Pico，它们提供基于上下文的,session(会话)的代理/拦截。

从 Hibernate3.0.1 版开始，加 Sessionfactory.getcurrentsession（）方法。最初此方法使用 JTA 的事务处理。在 JTA 自己的范围中与当前会话中一起使用。由此可以使用很成熟的 JTA TransactionManager（JTA 事务管理）实现。但是这要求所有应用程序都用 JTA 事务管理，不管这个应用程序是否部署在 J2EE 容器中，你的 session 一定要使用基于 JTA 的上下文。

然而，从 Hibernate3.1 以后，SessionFactory.getCurrentSession()可以变成插件式的，为此一个新的扩展接口 org.hibernate.context.spi.CurrentSessionContext 与一个新的配置参数 hibernate.current_session_context_class 产生了，通过这一变化，允许定义可插拔的上下文范围与当前会话。

参考 JAVA 文档中 org.hibernate.context.spi.CurrentSessionContext 接口，其中详细讨论了这一约定。接口定义了唯一方法 currentSession(),这个方法的实现类可以跟踪当前会话的上下文。在外部 Hibernate 允许用三种方法来实现这一接口：

org.hibernate.context.internal.JTASessionContext:由 JTA 事务界定当前会话范围与跟踪当前会话。这种处理方式与原来版本是完全一样。详情见 javadocs。

org.hibernate.context.internal.ThreadLocalSessionContext:由执行的线程跟踪当前会话。详情见 javadocs。

org.hibernate.context.internal.ManagedSessionContext:由执行的线程跟踪当前会话。然而你自己负责绑定与解绑 Session（会话）实例，这些通常在类的静态方法中，这时 Session（会话）不能打开，关闭，刷新，详情见 javadocs。

通常，这个参数的值将被命名为使用的实现类。当通过外部插件来实现，这些要分别要对应三个短语："jta","thread","managed"。

前两种实现都提供了“one session - one database transaction（一会话，一事务）”开发模式。这也可以称为*"session-per-request（每请求一会话）"*模式。开始结束一个 Hibernate 会话都是在数据库事务过程中定义的。如果计划使用经典的 JSE 平台的事务处理流程，而不使用 JTA 事务，建议在你的代码中用 hibernateTransaction（事务）API 来代替系统底层的事务过程。如果你使用 JTA，你可以利用 JTA 接口来定义事务。如果你运行在支持 CMT 的 EJB 容器中，事务以声明的方式定义，你的代码中将不用处理事务与会话的操作。参考第 6 章, *事务与并发控制*有更多的信息与代码示例。

hibernate.current_session_context_class 配置参数为 org.hibernate.context.spi.CurrentSessionContext。为了向后兼容，如果没有设定此参数，但是 org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform 被设定，hibernate 将使用 org.hibernate.context.internal.JTASessionContext。

# 第 2 章. Domain Model（域模型）

**目录**

术语 domain model（域模型）来自于 data modeling（数据建模）范畴。这个模型最终描述的是你要解决的 problem domain（问题域）。有时你听说过术语"持久化类（persistent classes）"。
归根结底，应用程序的 domain model（域模型）是实体关系模型（ORM）中的核心角色。他拼装你想要 map（映射）的类。Hibernate 可以非常好的工作于 POJO（Plain Old Java Object）或者 JavaBean 开发模式中。然而，这些规范都不是强制要求。事实上，Hibernate 没在限制持久层对象的特质。你可以用其它方式表示 domain model（域模型）（比如用 java.util.Map 的树状实例）。
注意

尽管 Hibernate 没有将这些规范看作是强制要求，但是 JPA 是要求的。因此如果你的应用有向 JPA 移植的可能性，你最好遵守严格的 POJO 模型。我们将指出在哪里涉及到这些问题。

本章将阐述 domain model（域模型）的特征，然而，不讨论 domain model（域模型）的全部内容。这是一个巨大的主题，它有自己的专用手册。参考 ***Hibernate Domain Model Mapping Guide（域模型映射手册）***，hibernate 文档页.

## 2.1. POJO 模型

本节探讨定义为 POJO 的域模型

### 2.1.1. 实现无参构造函数

POJO 要求实现一个无参构造函数，Hibernate 与 JPA 都要求这样。
JPA 要求这个构造函数声明为 public 或者 protected。Hibernate 大多数情况下只关心可见性，只要系统的安全机制允许重写可见性就可以。这意思是说，如果你想支持运行时代理生成类，这个构造函数应该声明最小包可见（译者注：默认作用域）。

### 2.1.2. 提供 identifier（标识）属性

注意

以前这被认为是可选的。然而在实体上不定义主键将被认为是过时的。在未来的版本中，将强制要求定义标识属性。

标识属性不一指要求是物理映射定义的主键列。然而，它一定映射到可以唯一标识一行的列上（译者注：唯一标识列）。
我们建议你在持久化类中声明一个统一命名的标识属性，你可以使用 nullable（可空的）类型（即：非 JAVA 基本数据类型）。

### 2.1.3. 使用非 final 类

Hibernate 核心功能是通过运行时代理可以懒加载（lazy load）实体数据。这个功能依赖于实体类是非终态类（final）或者 实现一个声明过所有属性 get 与 set 方法的接口。你可以坚持用终态类（final）并且不实现接口。但是你将不能使用代理进行懒加载关联抓取，并且这将限制你系统的优化选项。
注意

从 Hibernate5.0 开始，我们提供了一个强大的全新的字节码处理器，用于处理懒加载。以前版本的 hibernate 中只有一个简单的字节码重写功能。

你也应该避免持久化属性的 get,set 方法为终态的(fianl)的，原因上面已经说过了。

### 2.1.4. 为持久化属性声明 get,set 方法

虽然不是必须的，我们建议你遵守 JavaBean 规范来声明实体持久化属性的 get,set 方法。虽然 Hibernate 是可以直接访问实体的成员变量（field）。这此属性（成员变量，set,get 方法）不一定声明为 public。Hibernate 可以处理各种可见性（public, protected, package（译者注：默认属性）,private）的属性。

### 2.1.5. 实现 equals()与 hashCode()方法

注意

本节大部分内容讨论处理 Hibernate 会话（session）与实体的关系；如果你不熟悉实体的三种状态：manged（托管态）（译者注：原来叫 Persistent 持久态，这次改名了，还是有什么变化？），transient(瞬时态、临时态)，detached（脱管态、分离态），请参考这里

在域模型中是否实现 equals()与 hashCode()方法,如何实现他们。这一切在 ORM 中都是一个复杂的问题。

一般作为类中的标识类型，在其 id 值上必须实现 equals/hashCode 方法。有两种情况不一定要实现 equals/hashCode 方法，一是：当用户类使用联合主键时；二是：下面讨论的一些特殊情况。用户定义的类使用了复合键使用这种类用作定义用户类型的复合主键。除了以上这种情况以及下面讨论的很少一些情况下，你不必一定实现 equals/hashCode 方法。

那么，有什么烦恼？通常情况下，大多数的 Java 对象在其的标识属性上提供一个内置的 equals（）和 hashCode（）方法，因此，每个新的对象与其他对象不同。这通常是我们在普通的 Java 编程所要做的。然而，事实上当我们从数据库中取得数据时，很可能出现多个实例有相同的值。这时你应该开始考虑重定义内置的 equals（）和 hashCode（）方法。每次从数据库中加载一个特定下面代码的 Person 类，我们自然将得到唯一的实例。Hibernate 努力确保在一个会话中不发生这样的情况（译者注，多次加载同一行数据，对象应该是一个）。事实上，Hibernate 在一个特定内部会话中保证数据行的 ID 与 JAVA 的 ID 对等的关系。因此如果你在一个会话中请求一个特定对象多次，事实上你将得到同一个*实例*:

**示例 2.1. 标识符（ID）的作用域**

```
Session session = ...;
    Person p1 = session.get( Person.class, 1 );
Person p2 = session.get( Person.class, 1);

//assert p1 == p2; 的结果为 true
```

思考另一个例子中使用一个持久化的 java.util.Set:

**示例 2.2. Set 在会话作用域中的用法**

```
Session session = ...;

Club club = session.get( Club.class, 1 );

Person p1 = session.get( Person.class, 1 );
Person p2 = session.get( Person.class, 1);

club.getMembers().add( p1 );
club.getMembers().add( p2 );

// 以下计算结果为 true,也就是只加入了一个对象
 assert club.getMembers.size() == 1;
```

当从不同的会话加载时：

**示例 2.3. 混合会话**

```
Session session1 = ...;
```

```
Session session2 = ...;
Person p1 = session1.get( Person.class, 1 );
```

```
Person p2 = session2.get( Person.class, 1);

//计算结果为 false
assert p1 == p2;
Session session1 = ...;
Session session2 = ...;

Club club = session1.get( Club.class, 1 );

Person p1 = session1.get( Person.class, 1 );
Person p2 = session2.get( Person.class, 1);

club.getMembers().add( p1 );
club.getMembers().add( p2 );

//计算结果取决于：
```

```
        assert club.getMembers.size() == 1;
```

最后一个示例的具体计算结果取决于 Person 类是否实现 equals/hashCode 方法，如果是实现了，结果怎么样。
考虑另一种情况：

**示例 2.4. 瞬时态实体的赋值**

```
        Session session = ...;

        Club club = session.get( Club.class, 1 );

        Person p1 = new Person(...);
        Person p2 = new Person(...);

        club.getMembers().add( p1 );
        club.getMembers().add( p2 );

        //这个计算结果...再次取决
        assert club.getMembers.size() == 1;
```

在某些情况下，你是否在会话外处理实例（不管是瞬态与分离态），特别是你将在 java 集合中使用对象时，你们应该考虑实现 equals/hashCode 方法。
常用的初始化 equals/hashCode 方法是，使用实体的 ID 属性作为计算的基础：

**示例 2.5. 本地实现 equals/hashCode 方法**

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

                ...

                @Override
```

```
        public int hashCode() {
                return id != null ? id.hashCode() : 0;
        }

        @Override
        public boolean equals() {
                if ( this == o ) {
                        return true;
                }
                if ( !( o instanceof Person ) ) {
                        return false;
                }

                if ( id == null ) {
                        return false;
                }

                final Person other = (Person) o;
                return id.equals( other.id );
        }
}
```

事实证明，正如我们在上面的示例看到的那样，当向一个 SET 添加一个 Person 类的瞬态实例时，结果取决于是否实现现 equals/hashCode 方法。

**示例 2.6. 还是麻烦**

```
Session session = ...;
session.getTransaction().begin();

Club club = session.get( Club.class, 1 );

Person p1 = new Person(...);
Person p2 = new Person(...);

club.getMembers().add( p1 );
club.getMembers().add( p2 );

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

```
        // 下面代码结果为 false!
        assert club.getMembers().contains( p1 );
```

这里的问题是以下内容有冲突：1、使用生成器生成的标识符（id）;2、SET 集合的约定；3、equals/hashCode 方法的实现。也就是说，对象的 equals/hashCode 值作为 SET 集合的一部分不应该被改变，但是这里究竟发生了什么？因为 equals/hashCode 是以生成 ID(标识符)为基础计算的，直到 session.getTransaction().commit() 调用，都没有被赋值.
注意，这里涉及到的是生成的标识符（ID）。如果你使用已赋值的标识符就没有这个问题，假设标识符在加入到集合之前已经被赋值。
另一个选择是强制标识符被生成，并且在加入集合之前：

**示例 2.7. 强制生成标识符（ID）**

```
        Session session = ...;
        session.getTransaction().begin();

        Club club = session.get( Club.class, 1 );

        Person p1 = new Person(...);
        Person p2 = new Person(...);

        session.save( p1 );
        session.save( p2 );
        session.flush();

        club.getMembers().add( p1 );
        club.getMembers().add( p2 );

        session.getTransaction().commit();

        // 将得到 false!
        assert club.getMembers().contains( p1 );
```

但是这常常是不可行的。
最后，更好的方法是实现 equals/hashCode 方法，利用 natural-id（自然 ID）或者 business-key（业务主键）。

**示例 2.8. 更好的 equals/hashCode 使用 natural-id（自然 ID）**

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

                @NaturalId
                private String ssn;

                protected Person() {
                        // ctor for ORM
                }

                public Person(String ssn) {
                        // ctor for app
                        this.ssn = ssn;
                }

                ...

                @Override
                public int hashCode() {
                        assert ssn != null;
                        return ssn.hashCode();
                }

                @Override
                public boolean equals() {
                        if ( this == o ) {
                                return true;
                        }
                        if ( !( o instanceof Person ) ) {
                                return false;
                        }

                        final Person other = (Person) o;
                        assert ssn != null;
                        assert other.ssn != null;

                        return ssn.equals( other.ssn );
                }
```

```
        }
```

正如你看到的 equals/hashCode 的问题不是小事，也没有一种万能的解决方案。

## 2.2. Dynamic(动态)模型

持久化实体不一定用 POJO/JavaBean 这一种方式表示。Hibernate 也支持动态模型（用 Map 或者运行时 Map）。用这种方法，你不用改写持久化类，只是用映射文件。

注意

动态模型的映射已经超出了本文档的范围。我们只讨论在 Hibernate 中使用这些模型，但是有关映射的详情请参考《***Hibernate Domain Model Mapping***》。

一个指定的实体在一个指定的 SessionFactory 中只能有一种实体模型。但是这些在以前的版本中就已经更改了，现在允许为一个实体定义多个实体模型并且允许使用者选择如何加载。实体模型现在可以与域模型混合；动态实体可参考一个 POJO 实体，反之亦然。

**示例 2.9. 使用动态域模型工**

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

        // 建立一个 custormer 实体
        Map david = new HashMap();
        david.put("name", "David");

        // 建立一个 organization 实体
        Map foobar = new HashMap();
        foobar.put("name", "Foobar Inc.");

        // 连接两者
        david.put("organization", foobar);

        // 保存两者
        s.save("Customer", david);
```

```
        s.save("Organization", foobar);

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

动态模型主要的优点在于：快速的变化与运行，因为原型不用实体类实现。主要的缺点在于：你不能进行编译时检查，并且可能要处理大量的运行时异常。然而，做为 Hibernate 映射的结果，数据库 schema 可以更容易的标准化与合理化，并且以后在动态模型上添加更合适的域模型。
动态模型大量用于处理某些整合用例，例如：利用动态模型描述历史数据被广泛使用。

# 第 3 章 Bootstrap（引导、启动

**目录**



术语 *bootstrapping（引导）*是指初始化并且启动软件的组件。在 Hibernate 中我们将具体讨论建立全功能 SessionFactory 实例的过程或者在 JPA 中建立 EntityManagerFactory 实例的过程。这些过程各不相同。
注意
本章不是关注于所有引导的可能性。这些将包括在稍后的相关章节中。取而代之，我们关注于执行引导的 API 调用。

## 3.1. Native（原生、本地）引导

本节讨论 Hibernate SessionFactory 的引导过程。具体讨论 5.0 中重新设计的引导 API。对于 Legacy（旧版、过时）引导 API 的讨论，请参考 附录 A *Legacy（旧版、过时）引导*

### 3.1.1. 构建 ServiceRegistry

Native（原生、本地）引导的第一步是建立 ServiceRegistry，ServiceRegistry 持有 Hibernate 引导与运行时所需要服务。
事实上，我们涉及到建立 2 种不同的 ServiceRegistries。第一种是 org.hibernate.boot.registry.BootstrapServiceRegistry。BootstrapServiceRegistry 持有 Hibernate 引导与运行时所需要服务。这归结为 3 个服务：
org.hibernate.boot.registry.classloading.spi.ClassLoaderService -它控制 Hibernate 如何与类加载器的互动
org.hibernate.integrator.spi.IntegratorService-它控制管理与发现 org.hibernate.integrator.spi.Integrator 的实例。
org.hibernate.boot.registry.selector.spi.StrategySelector-它控制 Hibernate 如何处理各种约定策略的实现。这是一个功能强大的服务，但对它的全面讨论超出本手册的范围。
如果你没有特殊要求，可以使用 Hibernate 默认行为处理有关 BootstrapServiceRegistry 服务（这是经常发生的情况，特别是在 java SE 环境中），这样就可以跳过建立 BootstrapServiceRegistry 的过程。
如果你希望改变 BootstrapServiceRegistry 的构建，可以通过控制 org.hibernate.boot.registry.BootstrapServiceRegistryBuilder 类来实现：

**示例 3.1. 控制 BootstrapServiceRegistry 构建过程**

```
BootstrapServiceRegistryBuilder bootstrapRegistryBuilder
                        = new BootstrapServiceRegistryBuilder();
        // 添加一个自定义的类加载器
        bootstrapRegistryBuilder.applyClassLoader( mySpecialClassLoad
er );
        //或手工添加一个集成器（Integrator）
        bootstrapRegistryBuilder.applyIntegrator( mySpecialIntegrator
 );
```

```
        ...

        BootstrapServiceRegistry bootstrapRegistry =
bootstrapRegistryBuilder.build();
```

该 BootstrapServiceRegistry 的服务不能被继承（添加），也不能被重写（替换）。

第二种 ServiceRegistry 是 org.hibernate.boot.registry.StandardServiceRegistry.你将需要配置 StandardServiceRegistry，这是通过 org.hibernate.boot.registry.StandardServiceRegistryBuilder 来实现:

**示例 3.2. 构建 BootstrapServiceRegistry**

```
        // 这是一个隐式建立 BootstrapServiceRegistry 的例子
        StandardServiceRegistryBuilder standardRegistryBuilder
                    = new StandardServiceRegistryBuilder();

        // 这是一个显式建立 BootstrapServiceRegistry 的例子
        BootstrapServiceRegistry bootstrapRegistry = ...;

        StandardServiceRegistryBuilder standardRegistryBuilder
                    = new
StandardServiceRegistryBuilder( bootstrapRegistry );
```

一个 StandardServiceRegistry 是高度可配置的，这通过 StandardServiceRegistryBuilder 完成。参考关于 StandardServiceRegistryBuilder 的 javadocs 了解所有细节。一些相关的具体方法有：

**示例 3.3. 控制 StandardServiceRegistry 的构建过程**

```
StandardServiceRegistryBuilder standardRegistryBuilder = ...;

        // 通过资源查找加载某些属性（译者注：属性文件方式加载 ）
```

```
        standardRegistryBuilder.loadProperties( "org/hibernate/example/MyProperties.properties" );

        // 从 cfg.xml 配置文件来配置注册信息
        standardRegistryBuilder.configure( "org/hibernate/example/my.cfg.xml" );

        // 应用随机设置
        standardRegistryBuilder.applySetting( "myProp", "some value" );

        // 应用服务初始化器
        standardRegistryBuilder.addInitiator( new CustomServiceInitiator() );

        // 应用服务接口
        standardRegistryBuilder.addService( SomeCustomService.class, new SomeCustomServiceImpl() );

        // 最后建立 StandardServiceRegistry
        StandardServiceRegistry standardRegistry = standardRegistryBuilder.build();
```

### 3.1.2. 构建 Metadata

Native（原生、本地）引导的第二步是建立 org.hibernate.boot.Metadata 对象,这个对象包括程序域模型的解析结果并且这些结果映射到数据库中。显然第一步是需要找到这些被解析的信息源（带有注释的类（annotated classes），`hbm.xml` 文件，`orm.xml` 文件）。这一切的目的就是建立 org.hibernate.boot.MetadataSources:

**示例 3.4. 配置 MetadataSources**

```
MetadataSources sources = new MetadataSources( standardRegistry );

        // 做为一种选择，你可以不通过注册的服务来构建 MetadataSources，在这种情况下将建立一个默认的
        // BootstrapServiceRegistry，但上述方法是首选。
        // MetadataSources sources = new MetadataSources();
```

```
        // 添加一个类，其中使用了 JPA 或者 Hibernate 映射注释
        sources.addAnnotatedClass( MyEntity.class );

        // 添加一个类名，其中使用了 JPA 或者 Hibernate 映射注释。不同于上面的方法，上面的方法加载类时存在延时，
        // 如果使用 5.0 中新定义的运行时增强字节码这是很重要的改变。

        sources.addAnnotatedClassName( "org.hibernate.example.Customer" );

        // 添加命名的 hbm.xml 资源文件做为数据源：将查找 classpath 并且解析 XML 文件

        sources.addResource( "org/hibernate/example/Order.hbm.xml" );

        //添加 JPA 中的 orm.xml 资源文件做为数据源：将查找 classpath 并且解析 XML 文件

        sources.addResource( "org/hibernate/example/Product.orm.xml" );
```

MetadataSources 还有很多好方法；探索它们的 API 或者 javadoc 会得到更多的信息。此外，所有 MetadataSources 方法允许按你自己的风格进行链接：

**示例 3.5. 使用方法链（method chaining）配置 MetadataSources**

```
MetadataSources sources = new MetadataSources( standardRegistry )
                        .addAnnotatedClass( MyEntity.class )

        .addAnnotatedClassName( "org.hibernate.example.Customer" )

        .addResource( "org/hibernate/example/Order.hbm.xml" )

        .addResource( "org/hibernate/example/Product.orm.xml" );
```

一旦我们有了数据源映射信息定义，我们需要建立 Metadata 对象。如果你确定使用默认行为建立 Metadata，你可以很简单的调用 MetadataSources.buildMetadata()。

注意

注意，ServiceRegistry 可以在引导过程中的各个点之间进行传递。建议的处理方法是自己建立一个 StandardServiceRegistry，沿以下路径传递它：MetadataSources 构造函数，MetadataBuilder，Metadata，SessionFactoryBuilder，SessionFactory。这些对象都将持有同一个 StandardServiceRegistry。

如果你希望调整从 MetadataSources 建立 Metadata 的过程，你需要通过 MetadataSources#getMetadataBuilder 得到 MetadataBuilder。MetadataBuilder 允许很多种控制 Metadata 建立过程的方式。参考 javadoc 了解全部的细节。

**示例 3.6. 通过 MetadataBuilder 构建 Metadata**

```
MetadataBuilder metadataBuilder = sources.getMetadataBuilder();

        // 使用 JPA 兼容的隐式的命名策略
        metadataBuilder.applyImplicitNamingStrategy( ImplicitNamingSt
rategyJpaCompliantImpl.INSTANCE );

        // 当没有显示指定时，用表的 schema 名
        metadataBuilder.applyImplicitSchemaName( "my_default_schema" )
;

        Metadata metadata = metadataBuilder.build();
```

### 3.1.3. 构建 SessionFactory

native 引导的最后一步是建立 SessionFactory。像上面很多讨论一样，如果你确认使用默认行为从 Metadata 引用中建立 SessionFactory，你只要简单调用 Metadata#buildSessionFactory。

如果你希望调整建立过程，你需要通过 Metadata#getSessionFactoryBuilder 得到 SessionFactoryBuilder。请再次参考 javadoc 了解全部的细节。

**示例 3.7. 通过 SessionFactoryBuilder 建立 SessionFactory**

```
SessionFactoryBuilder sessionFactoryBuilder =
metadata.getSessionFactoryBuilder();

        // 提供一个 SessionFactory-level 拦截器
        sessionFactoryBuilder.applyInterceptor( new
MySessionFactoryInterceptor() );

        // 添加自定义观察器
        sessionFactoryBuilder.addSessionFactoryObservers( new
MySessionFactoryObserver() );

        // 应用 CDI BeanManager（Bean 管理器）（JPA 事件监听器）
        sessionFactoryBuilder.applyBeanManager( getBeanManagerFromSom
ewhere() );

        SessionFactory sessionFactory = sessionFactoryBuilder.build();
```

引导 API 是相当灵活的，但在大多数案例中最有意义的三个过程：

构建 StandardServiceRegistry

构建 Metadata

使用上面两个对象构建 SessionFactory

**示例 3.8. native 引导-完整代码**

```
StandardServiceRegistry standardRegistry = new
StandardServiceRegistryBuilder()
                        .configure( "org/hibernate/example/MyCfg.xml" )
                        .build();

        Metadata metadata = new MetadataSources( standardRegistry )
                        .addAnnotatedClass( MyEntity.class )

        .addAnnotatedClassName( "org.hibernate.example.Customer" )

        .addResource( "org/hibernate/example/Order.hbm.xml" )

        .addResource( "org/hibernate/example/Product.orm.xml" )
                        .getMetadataBuilder()
```

```
        .applyImplicitNamingStrategy( ImplicitNamingStrategyJpaCompli
antImpl.INSTANCE )
                        .build();

        SessionFactory sessionFactory =
metadata.getSessionFactoryBuilder()

        .applyBeanManager( getBeanManagerFromSomewhere() )
                        .build();
```

## 3.2. JPA 引导 JPA Bootstrapping

Hibernate 做为 JPA 提供者的引导可以用两种方式完成，一是 JPA 规范兼容模式；二是专用的引导过程。标准模式在某些环境中有很多的限制，除了这些限制外，JPA 标准模式是被大力推荐使用的。

### 3.2.1. JPA 兼容模式的引导

在 JPA 中引导最重要的是 javax.persistence.EntityManagerFactory 实例。JPA 规范定义 2 个主要标准引导途径，选择哪种途径，是根据应用程序计划如何访问 javax.persistence.EntityManager 实例，此实例从 EntityManagerFactory 中取得。我们为这两个途径定义了 2 个术语“EE”与“SE”，但是这两个术语很有迷惑性。JPA 规范的 EE 引导是指应用程序支持容器（EE，OSGi 等等）的情况，这些容器管理、注入持久化上下文。“SE 引导”是指其它的所有方式。我们在这个文档中用术语“容器引导（container-bootstrapping）”与“应用引导（application-bootstrapping）”来代表 EE 引导与 SE 引导。
如何你想了解关于使用 EntityManager 实例的更多细节,请参考 JPA2.1 说明书 7.6 与 7.7，那里分别涵盖了“container-bootstrapping（容器引导）”与“application-bootstrappin（应用引导）”。
为了兼容的 container-bootstrapping（容器引导），容器将为每个持久化单元建立 EntityManagerFactory。持久化单元的定义部署在 META-INF/persistence.xml 中，通过 javax.persistence.PersistenceUnit 注释或者 JDNI 查询注入到应用中。

**示例 3.9. 注入 EntityManagerFactory**

```
@PersistenceUnit
        EntityManagerFactory emf;
```

为了兼容 application-bootstrapping（应用引导），不用容器为应用建立 EntityManagerFactory，而是由应用程序自己建立 EntityManagerFactory。应用程序调用 javax.persistence.Persistence 类的 createEntityManagerFactory 方法建立实体管理器工厂：

**示例 3.10. 应用程序引导的 EntityManagerFactory**

```
        // 为“CRM”持久化单元建立一个 EMF（EntityManagerFactory）  .
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("CRM");
```

### 3.2.2. Proprietary 2-phase 引导

待办事宜

# 第 4 章 持久化 Context（上下文）

**目录**

org.hibernate.Session API 与 javax.persistence.EntityManager API 表示处理持久化数据的环境。这个概念称为持久化上下文（persistence context）。持久化数据有状态，这种状态表示底层数据库与持久化上下文之间的关系。

**实体状态**

transient（瞬时态） -实体刚刚被实例化并且没有与持久化上下文关联。也没有写入数据库中。通常还没有 ID 值。
managed（托管态）或者 persistent（持久态） -实体已经有了 ID 并且与持久化上下文关联。在物理数据库中可能存在也可能不存在。
detached（游离态） -实体已经有了关联的 ID，但是不再与持久化上下文关联（通常因为持久化上下文已经关闭或者实体被上下文 evicted(回收、清理））
removed（删除态） -实体已经有了 ID 并且与持久化上下文有了关联，然而数据库已经计划删除数据。
大部分 org.hibernate.Session 与 javax.persistence.EntityManager 的方法，是在这些状态之间移动对象。

## 4.1. 实体持久化

一旦你建立了一个实体的实例（使用标准的 new 操作符），它就是在一个 new 状态下。你可以将这个实体持久化，方法是与 org.hibernate.Session 或者 javax.persistence.EntityManager 关联。

**示例 4.1. 实体持久化的例子**

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

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

org.hibernate.Session 也有方法被命名为 persist。此方法严格遵循 JPA 规范中定义的 persist 方法。这个 org.hibernate.Session 中的方法是 javax.persistence.EntityManager 实现的代理。
如 DomesticCat 实体有一个生成的 ID，这个值在 save 或者 persist 时被赋值。如果 ID 没有自动赋值。应用程序在 save 或者 persist 之前就为实体 ID 赋上值了。

## 4.2. 删除实体

实体当然可以被删除。

**示例 4.2. 删除一个实体**

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

注意这很重要，Hibernate 自身能够删除游离态对象，但是 JPA 不允许。这说是说：org.hibernate.Session 通过 delete 方法可以删除托管态与游离态对象，但是 JPA 只能通过 javax.persistence.EntityManager 的 remove 方法删除托管态对象。

## 4.3. 获取没有初始化数据的实体

有时会涉及到 lazy loading（懒加载），从一个实体中得到一个引用，同时不加载他的数据是非常重要的能力。通常，这用于需要在两个实体间建立关联。

**示例 4.3. 获取没有初始化数据的实**

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

上面的代码是假设实体被定义为懒加载（一般通过运行时代理）。查看更多信息参考这里。有两种情况将抛出异常：1 是指定实体没有关联到实际数据库时；2 是应用程序试图访问返回代理的数据时。

## 4.4. 获取已经初始化数据的实体

这很普通，想要得到一个实体以及它的数据，如下所示：

**示例 4.4. 获取已经初始化数据的实体引用**

```
session.byId( Author.class ).load( authorId );
entityManager.find( Author.class, authorId );
```

在这些情况下，如果没有匹配的数据行被发现将返回 null。

## 4.5. 通过 natural-id（自然 ID）获得实体

除了允许通过标识符加载对象，Hibernate 还允许程序通过 natural identifier(自然 ID)加载对象。

**示例 4.5. 简单 natural-id（自然 ID）访问**

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

                @NaturalId
                String userName;

                ...
        }

        // 使用 getReference()方法建立关联（引用）
        Resource aResource = (Resource)
session.byId( Resource.class ).getReference( 123 );
        User aUser = (User)
session.bySimpleNaturalId( User.class ).getReference( "steve" );
        aResource.assignTo( user );


        // 使用 load()方法填充数据
        return
session.bySimpleNaturalId( User.class ).load( "steve" );
```

**示例 4.6. natural-id（自然 ID）访问**

```
import java.lang.String;

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

                @NaturalId
                String system;

                @NaturalId
                String userName;

                ...
```

```
        }

        // 使用 getReference()方法建立关联（引用）
        Resource aResource = (Resource)
session.byId( Resource.class ).getReference( 123 );
        User aUser = (User) session.byNaturalId( User.class )
                        .using( "system", "prod" )
                        .using( "userName", "steve" )
                        .getReference();
        aResource.assignTo( user );


        //使用 load()方法填充数据
        return session.byNaturalId( User.class )
                        .using( "system", "prod" )
                        .using( "userName", "steve" )
                        .load();
```

正如上面看到的，通过自然 ID（natural id）访问实体数据允许两种形式 load 和 getReference 。

在 Hibernate API 中通过标识符（identifier）和自然 ID（natural-id）访问持久化数据方法是一样的。它们定义了两种访问方式：

getReference

在此情况下，假设标识符一定是存在的，如何不存在就会报错。所以此方法一定不要用于测试是否存在。这是因为此方法关注建立与返回的代理是否与 Session 关联，而不是与数据库有关。此方法经典的用法是用于外键关联。

load

如果对象的标识符不存在将返回 null，如果存在就返回带有标识符的对象。

除了这两个方法，其它每个定义的方法都有一个 org.hibernate.LockOptions 参数。这种锁机制将在一个单独的章节讨论。

## 4.6. 刷新实体状态

你可以在任何时间 load（重新加载）实体的实例与实体的集合。

**示例 4.7. 刷新实体状态**

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

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

使用刷新的一种情况是：当知道数据库状态发生改时。刷新允许当前的数据库状态被重新写入实体实例与持久化上下文环境。
使用刷新的另一种情况是：使用数据库触发器初始化实体的某些属性时。请注意，除非指定关联的级联刷新，否则只有实体与实体集合被刷新（译者注：不定义级联刷新只刷新实体自身的属性，不包括关联的引用）。但是，请注意，Hibernate 有能力自动处理刷新。请参考本手册中关于非标识符生成属性（non-identifier generated attributes）的讨论。

## 4.7. 更改托管态或者持久态

在托管状态（持久状态）的实体可以被应用程序操纵，任何更改将被自动发现，并在持久化上下文刷新时持久化。这时不需要调用什么特殊的方法就可以使这些更改持久化了。

**示例 4.8. 更改托管状态**

```
Cat cat = session.get( Cat.class, catId );
        cat.setName( "Garfield" );
        session.flush(); //通常不必显示调用
Cat cat = entityManager.find( Cat.class, catId );
        cat.setName( "Garfield" );
        entityManager.flush(); //通常不必显示调用
```

## 4.8. 使用游离态数据

游离是一种工作过程，数据在持久化上下文范围之外。数据变成游离态有很多种方式。一但持久化上下文关闭，与之相关的所有数据都成为游离态。清理持久化上下文环境有同样的效果。从持久化上下文中回收特定的实体，可以使它

们变成游离态。最后，序列化也可以使反序列化的形式成为游离（原始实体还是托管态）。
游离后的数据仍然可以被操作，但是持久化上下文不再自动知道这些更改，应用程序需要进行干预才可以使这些更改持久化。

### 4.8.1. 复位游离态数据（游离态变成托管态）

复位是指游离态的实体实例被再次取回，并且再次与持久化上下文关联（再次变成托管态）。
重要的
JPA 不支持这种模式，只有 Hibernate 的 org.hibernate.Session 支持。

**示例 4.9. 复位游离态实体**

```
session.lock( someDetachedCat, LockMode.NONE );
session.saveOrUpdate( someDetachedCat );
```

这个方法的命名 update 可能会引用一些误导。这个方法不是意味 SQL UPDATE 将要马上执行。然而是意味着当持久化上下文刷新时 SQL UPDATE 可能被执行。但是在执行前，要经过查询（select）过程，这被称为 select-before-update。这是因为 Hibernate 更新前不知道以前的状态是否变化了，所以要进行比较。这种情况下，Hibernate 将从实际数据库中取得数据，然后决定数据是否需要更新。
针对游离的实体，update 和 saveOrUpdate 方法执行完全相同的操作。

### 4.8.2. 合并游离态数据

合并过程是指：将一个传入的游离态的对象中的数据复制到一个新的托管态的对象中。

**示例 4.10. 合并的演示代码**

```
Object detached = ...;
```

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

这不一定发生，但是这是一个很好的演示。

**示例 4.11. 合并游离态实体**

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

## 4.9. 验证对象的状态

程序能够验证持久化上下文中的有关实体与集合的状态。

**示例 4.12. 验证托管态**

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

**示例 4.13. 验证懒加载**

```
if ( Hibernate.isInitialized( customer.getAddress() ) {
                //如果加载就显示
        }
        if ( Hibernate.isInitialized( customer.getOrders()) ) ) {
                //如果加载就显示
        }
        if (Hibernate.isPropertyInitialized( customer,
"detailedBio" ) ) {
                //如果加载就显示属性
        }
```

```
javax.persistence.PersistenceUnitUtil jpaUtil =
entityManager.getEntityManagerFactory().getPersistenceUnitUtil();
        if ( jpaUtil.isLoaded( customer.getAddress() ) {
                //如果加载就显示
        }
        if ( jpaUtil.isLoaded( customer.getOrders()) ) ) {
                //如果加载就显示
        }
        if (jpaUtil.isLoaded( customer, "detailedBio" ) ) {
                //如果加载就显示属
        }
```

在 JPA 中用另一个模式检查懒加载：javax.persistence.PersistenceUtil。然而， javax.persistence.PersistenceUnitUtil 被推荐使用。

**示例 4.14. JPA 中验证懒加载**

```
javax.persistence.PersistenceUtil jpaUtil =
javax.persistence.Persistence.getPersistenceUtil();
        if ( jpaUtil.isLoaded( customer.getAddress() ) {
                //如果已经加载就显示 address
        }
        if ( jpaUtil.isLoaded( customer.getOrders()) ) ) {
                //如果已经加载就显示 orders
        }
        if (jpaUtil.isLoaded(customer, "detailedBio") ) {
                //如果已经加载就显示 detailedBio 属性
        }
```

## 4.10. 从 JPA 访问 Hibernate

JPA 定义了一个非常有用的方法，以方便程序访问底层提供者的 API。

**示例 4.15. 使用 EntityManager.unwrap**

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

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

# 第 5 章 访问数据库

**目录**



## 5.1. ConnectionProvider（连接提供器）

作为 ORM 工具，可能最重要的一件事就是告诉 Hibernate 如何连接你的数据库，同时这个连接也可能是你应用程序的连接。最基本的函数在 org.hibernate.engine.jdbc.connections.spi.ConnectionProvider 接口中。Hibernate 为此接口提供了一些外部的实现。ConnectionProvider 是一个扩展点，因此你可以自定义如何实现，不管是第三方插件，还是你自己编写。如何使用 ConnectionProvider 通过 hibernate.connection.provider_class 设置。了解细节请参考 org.hibernate.cfg.AvailableSettings#CONNECTION_PROVIDER。

一般面言，如果使用 Hibernate 提供的实现之一，应用程序就不用显式设置 ConnectionProvider。Hibernaet 自身会通过以下规则确定使用哪个 ConnectionProvider：

最高优先权是： hibernate.connection.provider_class。

第二是： hibernate.connection.datasource-> 5.1.1 节“使用 DataSource”

第三是 hibernate.c3p0. -> 5.1.2 节, “使用 c3p0”

第四是 hibernate.proxool. -> 5.1.3 节, “使用 Proxool”

第五是 hibernate.hikari. -> 5.1.4 节, “使用 Hikari”

第六是 hibernate.connection.url -> 5.1.5 节, “使用 Hibernate 内置（不支持）的 pooling”

其它 -> 5.1.6 节, “用户自定义的连接”

### 5.1.1. 使用 DataSources

Hibernate 可以整合 javax.sql.DataSource，从而得到 JDBC 连接。应用程序必须告诉 Hibernate 连接细节。这在 hibernate.connection.datasource 中定义（必须）。一般有两种方式，一是：JNDI 名称，二是引用一个真实的 DataSource 对象。JNDI 的用法参考 第七章 ,JNDI

注意

对于 JPA 应用，注意：hibernate.connection.datasource 对应

javax.persistence.jtaDataSource 或者

javax.persistence.nonJtaDataSource。

数据源的 ConnectionProvider（连接提供者）可以选择是否接受 hibernate.connection.username 和 hibernate.connection.password 参数。如果指定了参数，将调用 DataSource#getConnection 同时接收用户名与密码两个参数，否则使用无参的调用 。

### 5.1.2. 使用 c3p0

重要的

使用这种整合，应用程序必须将 hibernate-c3p0 的 jar 包（还有它的辅助包）放到 classpath 中。

Hibernate 还支持应用程序使用 c3p0 的连接池。当使用这些 c3p0 功能时，要进行一此额外的设置。

连接的事务隔离被 ConnectionProvider 自己管理。参考 5.1.7 节, “ConnectionProvider 事务隔离设置”。

**额外的设置**

hibernate.connection.driver_class
JDBC 驱动类名设置
hibernate.connection.url
JDBC 连接 url 路径
一些使用 hibernate.connection.做为前缀的设置（除了一些特殊项）。

这些 hibernate.connection. 前缀，在内部将被去除，剩余的部分传递给 JDBC 连接的属性文件（译者注：是指去除了前缀的剩余部分去比对 JDBC 属性文件中对应项）。

**hibernate.c3p0.min_size 或者 c3p0.minPoolSize**

c3p0 连接池的最小连接数。参考
http://www.mchange.com/projects/c3p0/#minPoolSize
**hibernate.c3p0.max_size 或 c3p0.maxPoolSize**

c3p0 连接池的最大连接数。参考
http://www.mchange.com/projects/c3p0/#maxPoolSize
**hibernate.c3p0.timeout 或 c3p0.maxIdleTime**

连接空闲时间，参考
http://www.mchange.com/projects/c3p0/#maxIdleTime
**hibernate.c3p0.max_statements 或 c3p0.maxStatements**

控制 c3p0 中 PreparedStatement 的缓冲大小（如果用到了）。参考
http://www.mchange.com/projects/c3p0/#maxStatements
**hibernate.c3p0.acquire_increment 或 c3p0.acquireIncrement**

当池用完时，c3p0 开启的连接数。参考
http://www.mchange.com/projects/c3p0/#acquireIncrement
**hibernate.c3p0.idle_test_period 或 c3p0.idleConnectionTestPeriod**

c3p0 连接池空闲连接的检测周期。参考
http://www.mchange.com/projects/c3p0/#idleConnectionTestPeriod
c3p0.initialPoolSize
c3p0 连接池初始连接数，如果没有的指定，就是最小连接值。参考
http://www.mchange.com/projects/c3p0/#initialPoolSize
**hibernate.c3p0.前缀的其它设置**

将 hibernate.这一部分去除，剩余的其它部分传递给 c3p0。
c3p0.前缀的其它设置

直接传递给 c3p0。参考
http://www.mchange.com/projects/c3p0/#configuration

### 5.1.3. 使用 proxool 连接池

重要的

使用用这种整合，应用程序一定包含 hibernate-proxool 的 jar 包（还有它的辅助包）放到 classpath 中。

Hibernate 也支持应用程序使用 Proxool 的连接池
连接的事务隔离被 ConnectionProvider 自己管理。参考 5.1.7 节，“ConnectionProvider 事务隔离设置”。

#### 5.1.3.1. 使用现有的 Proxool 连接池

由 hibernate.proxool.existing_pool 设置。如果为 true,ConnectionProvider 将通过别名使用一个已经存在的 Proxool 连接池。别名在 hibernate.proxool.pool_alias 中指定。

#### 5.1.3.2. 通过 XML 设置 Proxool

文件名为 hibernate.proxool.xml 的 Proxool 配置文件一定要包含在资源路径下，然后被 Proxool 的 JAXPConfigurator 加载。参考

http://proxool.sourceforge.net/configure.html。
hibernate.proxool.pool_alias 一定要有值，表示使用哪个连接池。

#### 5.1.3.3. 通过属性文件(Properties)设置 Proxool

文件名 hibernate.proxool.properties 的 Proxool 配置文件一定要包含在资源路径下，然后被 Proxool 的 PropertyConfigurator 加载。参考 http://proxool.sourceforge.net/configure.html。
hibernate.proxool.pool_alias 一定要有值，表示使用哪个连接池。

### 5.1.4. 使用 Hikari

重要的
使用用这种整合，应用程序一定包含 hibernate-hikari 的 jar 包（还有它的辅助包）放到 classpath 中。
Hibernate 也支持应用程序使用 Hikari 的连接池
所有的 Hikari 的设定都有 hibernate.hikari. 前缀。ConnectionProvider 将这些设定筛选出来并传递给 Hikari。另外，ConnectionProvider 筛选出来 Hibernate-specific 的属性，然后将这此选项映射到 Hikari 的对应项中（如果有冲突，hibernate.hikari. 前缀的选项有优先权）：

**Hibernate-specific 中被 Hikari 识别的属性**

hibernate.connection.driver_class
映射 Hikari 的 driverClassName 设定
hibernate.connection.url
映射 Hikari 的 jdbcUrl 设定
hibernate.connection.username
映射 Hikari 的 username 设定
hibernate.connection.password
映射 Hikari 的 password 设定
hibernate.connection.isolation
映射 Hikari 的 transactionIsolation 设定。参考 5.1.7 节, "ConnectionProvider 事务隔离设置"。注意 Hikari 仅支持 JDBC 标准隔离级别。

hibernate.connection.autocommit
映射 Hikari 的 autoCommit 设定

#### 5.1.5. 使用 Hibernate 内置的（不支持）的连接池

重要的

内置的连接池不可以使用

这部分在这里只是为了文档的完整性。

#### 5.1.6. 用户自定义的连接

当 Session 打开后，Hibernate 只是简单的传递连接给用户的 Session。这种做法是不鼓励的所以不在此讨论。

#### 5.1.7. ConnectionProvider 事务设置

所有提供的 connectionprovider 实现，除了 DataSourceConnectionProvider，都支持所有从底层连接池得到的事务隔离级别。
hibernate.connection.isolation 的值可以由 3 种格式指定：
JDBC（事务隔离）级别的整数值
java.sql.Connection 常数名，表示你想使用的事务隔离级别。例如：TRANSACTION_REPEATABLE_READ 是指
java.sql.Connection#TRANSACTION_REPEATABLE_READ。只支持标准的事务隔离，不是特定的 JDBC 驱动的事务隔离级别。
一个缩写名，表示 java.sql.Connection 的常量名，没有 TRANSACTION_前缀。例如，REPEATABLE_READ 是指
java.sql.Connection#TRANSACTION_REPEATABLE_READ。再次强调，这只是标准的事务隔离，不是特定的 JDBC 的驱动事务隔离级别。

### 5.2. 数据库 Dialect(方言)

虽然 SQL 比较规范，但是每个数据库供应商使用自己的 SQL 定义语法。这些语法是标准 SQL 的子集或者是超集。这就是数据库*方言（dialect）*的由来。

Hibernate 根据不同的数据库用不同的方言进行访问，这是通过 org.hibernate.dialect.Dialect 类与数据库供应商各种子类来完成的。

在大多数情况下，Hibernate 能够正确的选择适当的方言。这是通过在引导期与 JDBC 连接交互来实现的。关于 Hibernate 如何确定用什么方言（你自己也可以确定）的信息请参考 19.3 节, “Dialect（方言)解析”

如何由于一些原因不能确定是什么合适的方言或者是你想用自定义的方言，你需要设置 hibernate.dialect。

**表 5.1. 提供的 Dialect（方言）**

| Dialect（方言）（简写） | 备注 |
| --- | --- |
| Cache71 | 支持 CachÉ数据库，版本为 2007.1 |
| CUBRID | 支持 CUBRID 数据库 8.3 及以后版本 |
| DB2 | 支持 DB2 数据库 |
| DB2390 | 支持 DB2 通用数据库 OS/390，也可称为 DB2/390 |
| DB2400 | 支持 DB2 通用数据库 iSeries，也可称为 DB2/400 |
| DerbyTenFive | 支持 Derby 数据库 10.5 |
| DerbyTenSix | 支持 Derby 数据库 10.6 |
| DerbyTenSeven | 支持 Derby 数据库 10.7 |
| Firebird | 支持 Firebird 数据库 |
| FrontBase | 支持 Frontbase 数据库 |
| H2 | 支持 H2 数据库 |
| HSQL | 支持 HSQL(HyperSQL) |

| | 数据库 |
|---|---|
| Informix | 支持 Informix 数据库 |
| Ingres | 支持 Ingres 数据库 9.2 |
| Ingres9 | 支持 Ingres 数据库 9.3 及较新的版本 |
| Ingres10 | 支持 Ingres 数据库 10 及较新的版本 |
| Interbase | 支持 Interbase 数据库 |
| JDataStore | 支持 JDataStore 数据库 |
| McKoi | 支持 McKoi 数据库 |
| Mimer | 支持 Mimer 数据库 9.2.1 及较新版本 |
| MySQL5 | 支持 MySQL 数据库 5.x |
| MySQL5InnoDB | 支持 MySQL 数据库 5.x ，更适用于 InnoDB 引擎导出表 |
| MySQL57InnoDB | 支持 MySQL 数据库 5.7x 及较新版本 ，更适用于 InnoDB 引擎导出表, |
| Oracle8i | 支持 Oracle 数据库 8i |
| Oracle9i | 支持 Oracle 数据库 9i |
| Oracle10g | 支持 Oracle 数据库 10g |
| Pointbase | 支持 Pointbase 数据库 |
| PostgresPlus | 支持 Postgres Plus 数据库 |

| | |
|---|---|
| PostgreSQL81 | 支持 PostgrSQL 数据库 8.1 |
| PostgreSQL82 | 支持 PostgrSQL 数据库 8.2 |
| PostgreSQL9 | 支持 PostgrSQL 数据库 9 及较新版本 |
| Progress | 支持 Progress 数据库 9.1c 及较新版本 |
| SAPDB | 支持 SAPDB/MAXDB 数据库 |
| SQLServer | 支持 SQL Server2000 数据库 |
| SQLServer2005 | 支持 SQL Server2005 数据库 |
| SQLServer2008 | 支持 SQL Server2008 数据库 |
| Sybase11 | 支持 Sybase 数据库最高到 11.9.2 版 |
| SybaseAnywhere | 支持 Sybase Anywhere 数据库 |
| SybaseASE15 | 支持 Sybase Adaptive Server Enterprise database 数据库 15 版 |
| SybaseASE157 | 支持 Sybase Adaptive Server Enterprise database 数据库 17 及较新版本 |
| Teradata | 支持 Terdata 数据库 |
| TimesTen | 支持 TimesTen 数据库 5.1 及较新版本 |

# 第 6 章 事务与并发控制

**目录**



这是很重要的，事务有不同的含义对于持久化与 ORM（Object/Relational Mapping）。大部分情况下它们是一致的，但是不总是这样。
可能是指到数据库的物理事务概念。
可能是指与持久化上下文有关逻辑事务概念。
可能是指应用程序的工作单元（Unit-of-Work），作为原型定义时。
注意
本文档大部分讨论的是事务的物理与逻辑概念，并将它们看成是一样的。

## 6.1. 物理事务

Hibernate 使用 JDBC API 进行持久化。有两个定义好的机制处理 JDBC 事务：JDBC 与 JTA。Hibernate 支持这两种机制与事务的整合，同时允许应用程序自己管理物理事务。
事务处理是由每个会话的 org.hibernate.resource.transaction.TransactionCoordinator 处理。它由 org.hibernate.resource.transaction.TransactionCoordinatorBuilder 服务建立。TransactionCoordinatorBuilder 表示处理事务的策略，而 TransactionCoordinator 表示与会话相关的策略实例。使用哪个

TransactionCoordinatorBuilder 实现是由
hibernate.transaction.coordinator_class 设定。

### Hibernate 提供的 TransactionCoordinatorBuilder 实现

jdbc（默认）-通过调用 java.sql.Connection 管理事务
jta-通过 JTA 管理事务。参考这里
注意
有关用户自定义 TransactionCoordinatorBuilder 的细节，或者更好理解它如何工作请参考 ***Integrations Guide***

Hibernate 直接使用 JDBC 连接与 JTA 资源，不增加任何额外的锁定行为。Hibernate 不会锁定内存中的对象，当使用 Hibernate 时，你数据库的事务隔离行为不会有任何改变。Hibernate 会话只是做为事务作用域的缓存做两件事：1、提供可重复读取的查询标识符；2、在查询结果中加载实体。
重要的
在数据库中要减少锁竞争的情况，物理数据库事务要设计的越小越好。大型的数据库事务会阻碍应用程序升级，比如进行高并发的负载。不要在最终用户级（end-user-level）的工作中保留一个打开的事务，而要打开事务在最终用户级（end-user-level）工作完成后。这个概念就是：transactional write-behind（事务的延迟写、事务后写。

#### 6.1.1. JTA 配置

与 JTA 系统交互的是后台整合，命名为
org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform，它显示访问 javax.transaction.TransactionManager 与
javax.transaction.UserTransaction，然后注册
javax.transaction.Synchronization 实例，此实例可以检查事务状态等。
注意

通常 JtaPlatform（JTA 平台）处理 JTA TransactionManager，UserTransaction，等是通过访问 JNDI。参考第七章 *JNDI* 了解配置 JNDI 的细节。

Hibernate 想解析 JtaPlatform，需要使用另一个服务，名称为 org.hibernate.engine.transaction.jta.platform.spi.JtaPlatformResolver。如果这个解决方案不正常工作或者你想提供自定义的实现，你需要配置 hibernate.transaction.jta.platform。Hibernate 提供很多种 JtaPlatform 约定的实现，这都是通过短名称来实现的：

**通过短命名配置 JtaPlatform 实现**

Borland-Borland 的企业级服务
Bitronix-Bitronix 的 JtaPlatform
JBossAS-在 JBoss/WildFly 应用服务器中使用 Arjuna/JBossTransactions/Narnya
JBossTS-独立使用 Arjuna/JBossTransactions/Narnya
JOnAS-使用 JOnAS 的 JOTM
JOTM-独立使用 JOTM
JRun4-JRun4 应用服务器
OC4J-Oracle 的 OC4J 容器
Orion-Orion 应用服务器
Resin-Resin 应用服务器
SunOne-SunOne 应用服务器
Weblogic-Weblogic 应用服务器
WebSphere-老版本的 WebSphere 应用服务器
WebSphereExtended-新版本的 WebSphere 应用服务器

## 6.2. Hibernate 事务 API

Hibernate 提供一套 API 用于隔离应用对于底层事务的使用。基于 TransactionCoordinatorBuilder 的配置，当应用使用这些事务 API 时 Hibernate 可以简单地做出正确的判断。这此 API 还允许你的应用或是组件在不同平台之间迁移。

使用这些 API,你将从会话中得到 org.hibernate.Transaction
事务允许你所期望的所有操作。begin，commit，rollback。 甚至这些操作可以是空操作。
它甚至暴露出一些很酷的方法，像：

- markRollbackOnly 这个可以同时工作在 JTA 与 JDBC 中！
- getTimeout 与 setTimeout 再次可以被 JTA 与 JDBC 使用
- registerSynchronization 允许你注册 JTA 同步，甚至在非 JTA 环境中也可以。

事实上不管是 JTA 环境还是 JDBC 环境，这些同步都保存在 Hibernate 本地。

在 JTA 环境中，Hibernate 只是在 TransactionManager 中注册一个简单的同步，用于防止乱序的问题。

另外，它还暴露了 getStatus 方法，这个方法返回一个 org.hibernate.resource.transaction.spi.TransactionStatus 枚举。这个方法检查底层的事务系统，所以尽量少用，它可能对某些 JTA 性能产生很大的影响。
让我们看一下使用 Transaction API 中的各种环境。

**示例 6.1. JDBC 中使用**

```
        public void doSomeWork() {
                StandardServiceRegistry ssr = new
StandardServiceRegistryBuilder()//“jdbc”是默认的，但也可以显式指定

        .applySetting( AvailableSettings.TRANSACTION_COORDINATOR_STRA
TEGY, "jdbc" )                  ...;
                SessionFactory = ...;
                Session session = sessionFactory.openSession();
                try {
                        //一定设定了 Connection#setAutoCommit(false),
下面的方法开始事务
                        session.getTransaction().begin();
                        doTheWork();
                        // 调用 Connection#commit()，如果有错误，我们
尝试回滚
                        session.getTransaction().commit();
```

```
                        }
                catch (Exception e) {
                //我们可能会使用回滚，这要看发生了什么异常
                        if ( session.getTransaction().getStatus() ==
ACTIVE  || session.getTransaction().getStatus() == MARKED_ROLLBACK )
{
                                session.getTransaction().rollback();

                        }       //底层的错误处理
                }
                finally {
                        session.close();
                }
        }
```

**示例 6.2. JTA(CMT)（容器管理事务）中使用 Transaction（事务）API**

```
                public void doSomeWork() {
                        StandardServiceRegistry ssr = new
StandardServiceRegistryBuilder()

        .applySetting( AvailableSettings.TRANSACTION_COORDINATOR_STRA
TEGY, "jta" )
                                ...;
                                //注意：依赖 JtaPlatform 的一些可选设定，
底层事务通过 JTA TransactionManager 或 UserTransaction 控制。
                SessionFactory = ...;
                Session session = sessionFactory.openSession();
                try {
                        // 因为是在 CMT 中，JTA 事务已经启动了，这个调
用基本上是没有操作的。
                        session.getTransaction().begin();
                        doTheWork();
                        //              因为是在 CMT 中,我们不用结束它，
这个调用也是没用的。
                        session.getTransaction().commit();
                }
                catch (Exception e) {
```

```
                        //再次，这个回滚也没有操作，（除非标记为底层回
滚）

                        if ( session.getTransaction().getStatus() ==
ACTIVE  || session.getTransaction().getStatus() == MARKED_ROLLBACK )
{
                                session.getTransaction().rollback();
                        }
                        //处理底层的错误
                }
                finally {
                        session.close();
                }
        }
```

**示例 6.3. JTA(BMT)（bean 管理事务）使用 Transaction（事务）API**

```
                public void doSomeWork() {
                        StandardServiceRegistry ssr = new
StandardServiceRegistryBuilder().applySetting( AvailableSettings.TRAN
SACTION_COORDINATOR_STRATEGY, "jta" )...;
                        //注意：依赖 JtaPlatform 的一些可选设定，底层
事务通过 JTA TransactionManager 或 UserTransaction 控制。
                        SessionFactory = ...;
                        Session session = sessionFactory.openSession();

                        try {
                                //假设 JTA 事务没有激活，这将调用 TM/UT
的开始方法。如果 JTA 事务已经激活，如果 JTA 事务相关会话没有初始化，将
调用空操作的 commit 与 rollback。
                                session.getTransaction().begin();
                                doTheWork();
                                //调用 TM/UTcommit 方法，假设我们已经初
始化。
                                session.getTransaction().commit();
                        }
                        catch (Exception e) {
```

```
                                // 根据哪里发生了异常，我们可能需要回
滚。
                                if
( session.getTransaction().getStatus() == ACTIVE      ||
session.getTransaction().getStatus() == MARKED_ROLLBACK ) {
                                //  调用 TM/UT 的 commit 方法，假设我们
已经初始化。否则只标记 JTA 事务为 rollback。
                                session.getTransaction().rollback();
                                }
                                //处理底层的错误
                        }
                        finally {
                                session.close();
                        }
        }
```

在 CMT 中，我们完全可以忽略所有的事务调用。但是示例中的代码调用点只是为了展示代码与底层事务机制的隔离特性。 事实上，如果你去除注释与启动设置代码，3 个示例的代码是一样的，也就是说你可以为 3 个事务环境开发一套代码。

事务 API 力图所有环境下代码一致。为此，当有不同时通常使用 JTA 规范代码来实现（例如 提交失败时的回滚操作）。

## 6.3. 事务模式（与反模式）

### 6.3.1. Session-per-operation（每操作一个会话）反模式

在单线程中每次数据库访问都打开、关闭 Session，这是一种反模式（译者注：不好的模式，不推荐使用，反面典型）。同样对于数据库事务也一样是反模式。将你的数据库访问放到一个计划好的序列中。同样，程序的每条 SQL 语句不自动提交。Hibernate 希望应用服务器禁用自动提交模式。数据库事务不是可有可无的。所有与数据库的通信都封装在事务中。避免自动提交数据读取的行为，因为相对于明确定义的工作单元，很多小事务可能运行的不好，同时也不利于维护与扩展。

注意

使用自动提交不是不用数据库事务，相反当处于自动提交模式时，JDBC 驱动在每次请求后简单的执行一个隐式事务。这好象是你的程序每次 JDBC 调用后调用 commit 操作。

### 6.3.2. Session-per-request（每请求一个会话）模式

这是常用的事务模式。术语 request(请求)涉及到一个系统的概念，这个系统与客户端(用户)发生的一系列请求的交互。Web 应用是最典型的例子，但不是唯一的。在这种系统中，开始处理请求前，应用程序打开 Hibernate Session（会话），开始事务，处理所有与数据有关的工作，结束事务然后关闭会话。这种模式的核心就是事务与会话是一对一的关系。

在这种模式中，有一个通用的技巧，就是定义一个*当前会话（current session）*，用于简化操作。所有应用组件想要访问会话（Session），都可能直接访问这个*当前会话（current session）*。Hibernate 支持这个技术是通过 SessionFactory 的 getCurrentSession 方法。当前会话中的“当前”的概念核心就是找到当前的范围。这就是 org.hibernate.context.spi.CurrentSessionContext 的目的。它有两种有效的范围：

第一种：是 JTA 事务，因为 JTA 事务允许回调 hook，通过这个回调知道什么时候事务结果，这就给了 hibernate 关闭与清理 Session 的机会。这一过程是通过 org.hibernate.context.internal.JTASessionContext 完成的，它是 org.hibernate.context.spi.CurrentSessionContext 的实现类。通过这个实现，一个 Session 将在事务开始时，调用 getCurrentSession 方法时打开。

第二种：是应用程序请求周期自身。这一方法用 org.hibernate.context.internal.ManagedSessionContext 完成的，它是 org.hibernate.context.spi.CurrentSessionContext 的实现类。这里，一个外部组件负责管理“当前”会话的生命周期与作用域。这个作用域开始时，调用 ManagedSessionContext 的 bind（绑定）方法，并传入一个 Session.结束时，调用 unbind（解绑） 方法。

这些“外部组件”常用的示例包括：

javax.servlet.Filter（过滤器）的实现。

AOP 中的拦截器，它的注入点是服务层的方法。

代理或拦截容器。

重要的

在 JTA 环境下是，getCurrentSession()方法有一个缺点，如果你使用了它，那么 after_statement（语句后）连接释放模式（这一般是默认使用的）。由于 JTA 规范的限制，Hibernate 无法对 scroll()或 iterate()方法返回的 ScrollableResults 或者 Iterator 实例，进行自动清理与关闭。释放底层数据库的游标只能通过在 finally 块中显示调用 ScrollableResults.close()或者 Hibernate.close(Iterator)方法。

### 6.3.3. Conversations（对话）

session-per-request（每请求一会话）模式不是设计工作单元唯一的有效途径。很多业务流程需要一系列用户与数据库之间的交互。 在 WEB 与企业级应用中，不允许跨用户的数据库事务，思考以下的示例：

**程序 6.1 一个长对话的示例**

对话打开后的第一个界面，用户加载了一个特定的 Session 与事务。用户可以任意修改对象。
5 分钟后用户使用 UI 元素保存工作。修改结果被序列化。用户还希望在整个访问期间独占数据。
即使我们有多个数据库访问，从用户的角度来看，这一系统的步骤是一个单一的工作单元。在你的应用程序中我们有很多方法实现这一过程。
第一种初级实现是在用户编辑期保持 Session 与事务的打开状态。使用数据库级锁阻止其它用户修改相同的数据，保持数据的隔离性与原子性。这是一种典型的反模式，因为锁争是瓶颈，这将妨碍未来的扩展性。
几个数据库事务实现一个对话。在这各情况下，保持业务过程的隔离性是成为应用层的责任。一个单独的对话通常跨跃多个数据库事务。如果只有一个数据库事务（通常是最后一个）可以保存、更新数据，那么这些数据库访问可以成为一个原子。其它的事务只能是只读的。通常是通过一个向导式的对话框来接

收这些数据，这时可能跨跃多个请求与响应。hibernate 包含一些特性可以很方便的实现这一过程。

| | |
|---|---|
| 自动版本控制 | Hibernate 可以执行自动乐观锁并发控制。它能够自动检测用户等待时间（think time）的并发修改。检查这些是在对话结束时。 |
| Detached Objects（游离对象） | 如果你决定使用 每请求一个会话的模式，在用户等待时间中，所有加载的实例将处于游离状态。Hibernate 允许你再次连接对象或者序列化对象的修改。这种模式叫游离对象的每请求一个会话（session-per-request-with-detached-objects），自动版本控制用于隔离并发修改。 |
| Extended Session（延伸会话） | Hibernater 的 Session 可以在数据库事务提交后关闭与底层 JDBC 连接，当新的客户请求来到后再次打开连接（译者注：还是刚才关闭的 session）。这种模式称为 session-per-conversation（每对话一个会话）。自动版本控制用于隔离当前的修改与会话，会话不能自动刷新，只能显示操作。 |

Session-per-request-with-detached-objects 与 session-per-conversation 都有各自的优点与缺点。

#### 6.3.4. Session-per-application （每应用一个会话）

以后讨论

## 6.4. 常见问题

Session 不是线程安全的，思考下面并发内容：如 HTTP 请求，session bean，Swing workers。如果 Session 是共享状态，这将导致它们之间的冲突。如果你在 javax.servlet.http.HttpSession 中持有 Hibernate Session，你应该考虑对 HttpSession 使用同步操作；否则一个用户快速点击提交操作，可能会一个会话中并发多个线程。

Hibernate 抛出异常意味你要回滚数据库事务，并且立即关闭 Session。如果你的 Session 绑定了 Application（应用），你要停止应用。回滚数据库事务操

作不是将你的业务对象恢复到事务开始时的状态。这意味着数据库状态与业务状态不同步。通常这不是问题，因为异常是不可恢复的，回滚后你要重新开始你的操作。

Session 缓存持久状态下的每个对象（hibernate 监听与检查它们的改变）。如果你打开 Session 时间过长或者有加载了大量数据。缓存将不断增长直到抛出 OutOfMemoryException（内存越界异常，内存不足）。一种解决方案是：调用 clear()与 evict()方法来管理 Session 缓存。但是你也应该考虑一些替代处理大量数据的方法，如存储过程。Java 没有为这些操作提供合适的工具。一些解决方案参考 第 10 章 批处理。在用户操作期间保持 Session 打开也意味着有很高概率出现脏数据。

## 第 7 章 JNDI

Hibernate 可选通过 JNDI 交互。一般当应用在：

请求 JNDI 绑定的 SessionFactory

指定 JNDI 命名的 DataSource

所用的 JTA 事务和 JTA 平台需要通过 JDNI 查找 TM,UI 等

所有这些 JNDI 调用通过一个单一服务

org.hibernate.engine.jndi.spi.JndiService。这是一个标准 JNDI 服务，通过下面的参数配置：

hibernate.jndi.class-指定类名，此类是 javax.naming.InitialContext 的实现类。参考 javax.naming.Context#INITIAL_CONTEXT_FACTORY

hibernate.jndi.url-指定 JNDI 初始化上下文的连接 URL。参考 javax.naming.Context.PROVIDER_URL

其它以 hibernate.jndi.为前缀的设置将直接传递给 JNDI 提供者

注意

标准 JNDI 服务假设所有的 JNDI 调用针对同一个 InitialContext。如果你的应用使用多个命名服务器，你需要自定义 JndiService 的实现类来处理这些细节。

## 第 8 章 锁

**目录**

锁是指关系型数据库中的数据在被读取与使用期间不被改变的操作。
你的锁策略可以是*乐观锁（optimistic）*或者*悲观锁（pessimistic）*。

**锁策略**

乐观锁
乐观锁假设多个事务正常完成非没有相互影响，因此在事务提交之间，事务运行时不锁定它们影响的数据资源。提交之间，每个事务效验是否有其它事务修改数据。如果有冲突的修改，这个提交的事务就回滚[1]。
悲观锁

悲观锁假设并发事务之间有冲突，要求它们读取数据后就锁定资源，直到应用完成，不再使用数据才解除锁定。

Hibernate 提供机制在你的应用程序中实现这两种类型的锁。

## 8.1. 乐观锁

当你的应用程序使用长事务或者对话跨跃多个数据库事务时，你能保存带有版本的数据。因为如果同一个实体被两个对话更新，最后提交的更改被通知有冲突，并且不会覆盖另一个对话的工作。这种方法保证了隔离性，同时在*（Read-Often Write-Sometimes（经常读，偶尔写）*的环境下有很好的粒度与工作效率。
Hibernate 提供两种机制保存版本信息，一种是专用版本号，一种是时间戳。
版本号（Version number）

时间戳（Timestamp）

注意

对于一个游离态实体，版本号或者是时间戳永远不可为空。如果 Hibernate 检测到任何实体有空的版本号与时间戳，不管是否有未保存的值与你指定的策略

是什么，Hibernate 将此实体看成瞬态实体。声明一个空版本号或时间戳是一种简便的方法，用于防止在 Hibernate 中的过度的传递依赖。特别是在你使用指定的标识符或是联合主键时。

### 8.1.1. 指定版本号

乐观锁中，应用版本号机制是通过@Version 注释。

**示例 8.1 @Version 注释**

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

这里，版本属性被映射到 OPTLOCK 列上，实体管理器使用此列检测更新冲突，防止更新被覆盖的损失，这也就防止 *last-commit-wins（最后提交成功）*策略成立（译者注：先提交的成立，后提交不成立）。

版本列可能是任意数据类型，只要你定义并且实现适当的 UserVersionType 就可以了。

你的应用程序不能更改由 Hibernate 设定的版本号。想要人为递增版本号（译者注：版本号一般就是一个递增整数），请参考 Hibernate 实体管理参考文档中讲解的 LockModeType.OPTIMISTIC_FORCE_INCREMENT 或者 LockModeType.PESSIMISTIC_FORCE_INCREMENTcheck 属性。

数据库生成版本号

如果版本号由数据库建立，好象一个触发器，使用注释 @org.hibernate.annotations.Generated(GenerationTime.ALWAYS)。

**示例 8.2　在 hbm.xml 中声明一个版本属性**

```
        <!--
            ~ Hibernate, Relational Persistence for Idiomatic Java
            ~
            ~ License: GNU Lesser General Public License (LGPL), version 2.1 or later.
            ~ See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.
            -->
        <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|."
        />
```

| | |
|---|---|
| column | 指定代表版本号的列名，可选，默认为属性名 |
| name | 持久化类中的属性名 |
| type | 版本号类型，可选，默认 integer |
| access | Hibernate 访问属性值的策略，可选，默认 property |
| unsaved-value | 表示实例是新建并且没有保存。这区别与以前会话建立与保存过的游离实例。默认值 undefined，表示需要使用标识属性的值，可选。 |
| generated | 表示版本属性值被数据库建立。可选，默认 never。 |
| insert | 是否在 SQL 的 insert 语句中包含 version（版本）列。默认值 true，但是你可以设为 false,前提是数据库列已经定义并默认值为 0(译者注：版本列有默认值 0 时才能将此项定义为 false)。 |

### 8.1.2. Timestamp（时间戳）

在乐观锁中时间戳相对于版本号来说是不可靠的，但是它可以有其它一些用途。时间戳是自动使用的，如果你将注释@Version 加在 Date 或 Calendar 类型的属性上，它就自动定义为时间戳。

**示例 8.3. 在乐观锁中使用时间戳**

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

Hibernate 可以通过检索@org.hibernate.annotations.Source 注释的值来决定从数据库还是 JVM 取得时间戳。这个值可以是 org.hibernate.annotations.SourceType.DB（数据库取值）或 org.hibernate.annotations.SourceType.VM（虚拟机取值）。默认是从数据库中取。你如果不定义注释也是从数据库中取。
如果你使用注释 @org.hibernate.annotations.Generated(GenerationTime.ALWAYS)，时间戳也是由数据库产生而不是由 Hibernate 产生。

**示例 8.4. hbm.xml 中时间戳元素**

```
        <!--
            ~ Hibernate, Relational Persistence for Idiomatic Java
            ~
            ~ License: GNU Lesser General Public License (LGPL), version 2.1 or later.
            ~ See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.
            -->
        <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|."
        />
```

| | |
|---|---|
| column | 指定持有时间戳的列名，可选，默认与属性同名。 |
| name | 持久化类中 Java 类型为 Data 或者 Timestamp 的属性名称 |
| access | Hibernate 访问属性值的策略，可选，默认 property |
| unsaved-value | 表示实例是新建并且没有保存。这区别与以前会话建立与保存过的游离实例。默认值 undefined，表示需要使用标识属性的值，可选。 |
| source | Hibernate 选择从数据库还是当前的 JVM 检索时间戳。基于数据库的时间戳都会造成开销，因为 Hibernate 需要每次都向数据为发查询请求，以确认下次增量的值。然而从数据库中得到时间戳在集群环境下是相对安全的。不是所有的数据库支持检索当前时间。另外也可能是不安全的锁，因为会丢失精度（译者注：不同数据库的时间精确不同）。 |
| generated | 选择时间戳的值是否由数据库生成，可选 ，默认是 never（译者注：不用数据库生成） |

## 8.2. 悲观锁

通常，你只需要指定一个 JDBC 连接的隔离级别然后由数据库处理锁的问题。如果你需要为一个新开始的事务得到一个专用的悲观锁或者 再次恢复原来的锁，Hiebernate 提供你所需的工具。

注意

Hibernate 总是对数据库使用锁机制，从不锁定内存中的对象。

### 8.2.1. LockMode 类

LockMode 类定义不同的锁级别，这些由 Hibernate 获得。

| | |
|---|---|
| LockMode.WRITE | 当 Hibernate 更新或插入行时自动获得 |
| LockMode.UPGRADE | 用户显式请求 SELECT ... FOR UPDATE，当数据库支持这种语法时得到锁。 |
| LockMode.UPGRADE_NOWAIT | 用户显式请求 SELECT ... FOR UPDATE NOWAIT，在 ORACLE 中获得锁。 |
| LockMode.UPGRADE_SKIPLOCKED | 用户显式请求在 Oracle 中，用语法 SELECT ... FOR UPDATE SKIP LOCKED 或者有 SQL Server 中用语法 SELECT ... with (rowlock,updlock,readpast)，获得锁。 |
| LockMode.READ | 当 Hiberanet 的隔离级别为 Repeatable Read（重复读）或 Serializable（序列化）时自动获得锁。这种锁可以通过用户的显式请求重复获取。 |
| LockMode.NONE | 没有锁。所有对象在事务结束时切换到这种锁模式下。会话关联的对象调用 update()或 saveOrUpdate()方法时也开始进入这种锁定模式。 |

上面提到的用户显式请求会造成下面一些后果

调用 Session.load()，指定锁模式。

调用 Session.lock()。

调用 Query.setLockMode()。

如果你选择了 UPGRADE, UPGRADE_NOWAIT 或者 UPGRADE_SKIPLOCKED 级别，并调用 Session.load()，当请求的对象还没有被 session 加载，对象将使用 SELECT ... FOR UPDATE 加载。如果你调用 load()加载的对象，这个对象已经被低级的锁锁定了，Hibernate 调用 lock()方法锁定对象（译者注：重新用高级锁设置对象）。

如果指定锁模式是 READ,UPGRADE,UPGRADE_NOWAIT 或者 UPGRADE_SKIPLOCKED，Session.lock()执行版本号检查。在这种情况下，使用

UPGRADE,UPGRADE_NOWAIT 或者 UPGRADE_SKIPLOCKED, SELECT ... FOR UPDATE 语法。
如果数据库不支持请求的锁模式，Hibernate 使用一个合适的替代模式，而不是抛出一个异常。这保证了应用程序的可移植性。

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

# 第 9 章 Fetching（抓取）

**目录**



抓取，本质上，就是从数据库中取得数据的过程，这些数据被程序使用。调整应用程序如何抓取数据是确定应用如何执行的一个很大的因素。抓取太多的数据，比如在宽度（值、列）与/或深度（结果、行）方面，会增加不必要的开销。这此开销主要来源于 JDBC 通信与结果集处理。抓取太小的数据，导致导致额外的抓取操作。调整应用程序如何抓取，是一个影响应用整体性能的好机会。

## 9.1. 基础

抓取的概念分解成两个不同的问题。
什么时候抓取数据？现在？将来？
数据如何抓取？
注意

“现在”通常被称为 eager 或者 immediate（译者注：也就是立即抓取）。

“将来”通常被称为 lazy 或者 delayed（译者注：也就是延迟抓取）。

有很多个范畴定义抓取：

- *静态抓取*-静态定义的抓取策略，是由 Mappings(映射文件)完成的。静态抓取策略用于没有定义动态抓取策略时。 [2].

*动态抓取* （有时也称为运行时抓取）-动态定义是以真实的用例为中心的，有 2 种途径定义动态抓取：
*抓取配置文件* -定义在映射文件中，但是可以在会话中启用或禁用。
HQL/JPQL 与 Hibernate/JPA Criteria 查询有指定抓取的能力。具体说就是查询。
从 Hibernate4.2(JPA 2.1)后，你能够使用 JPA EntityGraphs。

**策略**

SELECT
执行独立的 SQL select 查询语句加载数据。这时既可以 EAGER（第二个 select 马上执行）或者 LAZY（第二个 select 延迟执行直到需要相关的数据时），这个被称为 N+1 次查询。（译者注：hibernate 的查询机制是先查询行的标识符，然后再按标识符查询每行的具体内容。所要查询 N 行，一定会执行 N+1 行的，多的那行就是查询标识符的语句）
JOIN
始终使用 EAGER 风格抓取。抓取数据通过使用 SQL 的 join 语句获得。（译者注：使用这个策略是所有查询都是一行 select 语句，所以不可能有延迟的情况出现。）
BATCH（批处理）
执行 Sql 独立的 Select 语句加载一部分相关的数据条目，条目的数据由 Sql 的 where 子句限定。同样，既可以 EAGER（第二个 select 马上执行）或者 LAZY（第二个 select 延迟执行直到需要相关的数据时）。（译者注：原文不是很明确，好像是说 hibernate.jdbc.fetch_size 属性的问题，不过这个属性不是通用的，mysql 就不支持）
SUBSELECT（子查询）
执行 SQL 的 select 语句的关联加载关联的数据，SQL 的限定由加载所有者提供。同样，既可以 EAGER（第二个 select 马上执行）或者 LAZY（第二个 select 延迟执行直到需要相关的数据时）。

## 9.2. 应用抓取策略

让我们考虑这些主题，这涉及到一个简单的域模型与几个用例。

**示例 9.1. 简单域模式**

```
@Entity
        public class Employee {
                @Id
                private Long id;

                @NaturalId
                private String userid;

                @Column( name="pswd" )
                @ColumnTransformer( read="decrypt(pswd)"
write="encrypt(?)" )
                private String password;

                private int accessLevel;

                @ManyToOne( fetch=LAZY )
                @JoinColumn
                private Department department;

                @ManyToMany(mappedBy="employees")
                @JoinColumn
                private Set<Project> projects;

                ...
        }
@Entity
        public class Department {
                @Id
                private Long id;

                @OneToMany(mappedBy="department")
                private List<Employees> employees;

                ...
        }
@Entity
```

```
public class Project {
        @Id
        private Long id;

        @ManyToMany
        private Set<Employee> employees;

        ...
}
```

重要的

Hibernate 推荐使用静态标记用于相关的 lazy（懒加载），使用动态抓取策略用于 eager（立即加载）。很遗憾，这不符合 JPA 规范，因为在 JPA 规范中要求所有 one-to-one and many-to-one 关联默认为 eager（立即加载）。Hibernate 作为 JPA 提供者尊重这种默认。

### 9.2.1. 不抓取

登录用例

作用第一个用例，考虑应用程序中雇员（Employee）的登录过程，假设登录过程中只需要访问雇员的个人信息，没有项目（Project）与部门（Department）信息。

**示例 9.2. 不抓取的示例**

```
String loginHql = "select e from Employee e where e.userid = :userid
and e.password = :password";
        Employee employee = (Employee) session.createQuery( loginHql )
                        ...
                        .uniqueResult();
```

在此示例中，应用程序只得到了雇员的个人数据。然而因为所有雇员的关联属性声明为 LAZY（JPA 默认集合为 LAZY，译者注：这里指的是 projects 属性），所以没有其它被数据抓取。
如果登录过程不需要访问雇员类的细节，另一种优化的抓取就是限制结果集宽度。

**示例 9.3. 不抓取（scalar）示例(译者注：只返回数值不是对象)**

```
String loginHql = "select e.accessLevel from Employee e where
e.userid = :userid and e.password = :password";
        Employee employee = (Employee) session.createQuery( loginHql )
                        ...
                        .uniqueResult();
```

### 9.2.2. 通过查询动态抓取

雇员参与项目的用例

作为第二个用例，请考虑界面显示一个雇员所参与的项目。当然访问雇员是必须的，同时雇员对象的项目集合也是需要的。 部门信息，其它雇员或者其它项目信息就不需要了。

**示例 9.4. 动态查询抓取**

```
String userid = ...;
        String hql = "select e from Employee e join fetch e.projects
where e.userid = :userid";
        Employee e = (Employee) session.createQuery( hql )
                        .setParameter( "userid", userid )
                        .uniqueResult();

String userid = ...;
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Employee> criteria =
cb.createQuery( Employee.class );
```

```
        Root<Employee> root = criteria.from( Employee.class );
        root.fetch( Employee_.projects );
        criteria.select( root );
        criteria.where(
                cb.equal( root.get( Employee_.userid ),
cb.literal( userid ) )
        );
        Employee e =
entityManager.createQuery( criteria ).getSingleResult();
```

在此示例中，我们从一个简单查询中得到一个雇员与他参与的项目， HQL 查询与 JPA Criteria 查询都可以表现这个示例。在这两种情况下，都被解析成一个完整的查询然后得到所有信息。

### 9.2.3. 通过配置文件动态抓取

**工程中的 employee 用例使用了 natural-id**

假定我们想要改用 natural-id 的加载得到"雇员参与项目"的用例信息。通过 natural-id 加载，使用静态定义的抓取策略，但是不能暴露定义的抓取方法。所以我们利用抓取配置文件。

**示例 9.5. 抓取配置文件示例**

```
@FetchProfile(
                name="employee.projects",
                fetchOverrides={
                        @FetchOverride(
                                entity=Employee.class,
                                association="projects",
                                mode=JOIN
                        )
                }
        )
String userid = ...;
        session.enableFetchProfile( "employee.projects" );
        Employee e = (Employee)
session.bySimpleNaturalId( Employee.class )
```

```
                .load( userid );
```

这里雇员类通过 natural-id 得到，雇员的项目数据被立即抓取。如果雇员数据已经被缓冲解析过，那么项目数据自己单独解析。而当雇员数据没有被缓冲解析时，那么雇员与项目数据通过 Sql 查询的 join 语句一起解析，如我们上面所看到的。

[2] 除了 HQL/JPQL 的情况，其它参考 xyz。

# 第 10 章 批处理

**目录**



首先我们要决定这里要讨论什么？这有这么多概念。难道我们都要讨论？
JDBC 批量更新？
Session 的增量刷新？
无状态会话？
JAVA EE 批处理？
所有上面的内容？
其它？

## 10.1. JDBC 批处理

JDBC 提供批处理支持，就是多个 SQL statement 一起被表示为一个单一的 PreparedStatement（预处理指令）。这通常意味着驱动在一次通信中发送多个操作给服务器，这将节省对于数据库网络访问。Hibernate 可以利用 JDBC 批处理。下面这些设置控制批处理行为。
hibernate.jdbc.batch_size-在请求驱动执行批处理之前，控制 Hibernate 处理批处理指令的最大数量。零或负数禁用此特性。
hibernate.jdbc.batch_versioned_data-当批处理执行时，一些 JDBC 驱动可能返回错误的行数。如果你的 JDBC 驱动是这一类驱动，这项设置应为 false。另外，如果 JDBC 驱动允许 Hibernate 对版本数据使用批量 DML 并且允许

Hibernate 使用乐观锁检查返回的行数，那么它是安全的。目前为了安全，默认 false。（译者注：此项目的是保证 executeBatch()可以返回正确的行数，但是有此 JDBC 驱动不支持或者可能返回错误的行数，那么想安全得返回正确的行数，要保证如下内容：1 支持 DML 批处理，2 支持锁返回行。当驱动不支持时，设定此项为 false 就可以了，同时也就无法使用返回行的特性了）

hibernate.jdbc.batch.builder-自定义管理批处理功能的实现类类名。替换 Hibernate 的默认实现，不是个好主意。但是如果你希望这样做，那么这里设置一个自定义的类名，这个类是 org.hibernate.engine.jdbc.batch.spi.BatchBuilder 的实现类。

hibernate.order_update-强制 Hibernate 订阅 SQL 更新实体类型和主键值。这将导致更多的批处理发生，好处是减少高并发下的死锁机会，坏处是带来性能的下降，所以要如何选择，要看对你的应用到底是有好处还是有坏处。

hibernate.order_inserts-强制 Hibernate 订阅插入值。这将导致更多的批处理发生，好处是减少高并发下的死锁机会，坏处是带来性能的下降，所以要如何选择，要看对你的应用到底是有好处还是有坏处。

# 第 11 章 缓冲

**目录**



## 11.1. 配置二级缓存

hibernate 有集成第三方数据缓冲插件的能力，这些插件由第三方提供，并且工作于 Session 上下文之外。 本节明确指出如何通过设置来控制这一行为。

### 11.1.1. RegionFactory（注册工厂）

org.hibernate.cache.spi.RegionFactory 设定 Hibernate 与第三方缓冲插件的集成。hibernate.cache.region.factory_class 用于声明使用哪个插件。Hibernate 同时支持 2 种流行的缓冲库：Ehcache 与 Infinispan。

#### 11.1.1.1. Ehcache

重要的

Hibernate 内置集成了 Ehcache，如果要使用它，需要 hibernate-ehcache 模块的 jar 包（与此 jar 包所需的相关依赖库）加入 classpath 中，

hibernate-ehcache 模块定义 2 个具体的提供者：

```
ehcache
ehcache-singleton
```

#### 11.1.1.2. Infinispan

重要的

Hibernate 内置集成了 Infinispan，如果要使用它，需要 hibernate-Infinispan 模块的 jar 包（与此 jar 包所需的依赖库）加入 classpath。

hibernate-infinispan 模块定义了两个提供者：

```
infinispan
infinispan-jndi
```

### 11.1.2. 缓冲行为

除了指定提供者的设置，Hibernate 中还集成了控制各种缓冲行为的设置选项：**hibernate.cache.use_second_level_cache**-启用与禁用所有二级缓冲。默认为 true(启用)

- **hibernate.cache.use_query_cache**-启用与禁用查询结果的二级缓冲。默认为 false(禁用)
- **hibernate.cache.query_cache_factory**-查询结果缓冲脏数据的处理规范是脏数据无效。默认实现是不允许有脏数据。在应用程序中使用这个设定将原来

的规范解梆。这里指定实现了 org.hibernate.cache.spi.QueryCacheFactory 类的自定义类类名。

- **hibernate.cache.use_minimal_puts**-优化二级缓冲的操作，在频繁读时减少写操作。提供者通常设置为合适的值。
- **hibernate.cache.region_prefix**-定义一个名称，用于所有二级缓冲区域名的前缀。
- **hibernate.cache.default_cache_concurrency_strategy**-在 Hibernate 中二级缓冲区可以设定不同的并发访问策略。这个设置允许定义默认的并发访问策略。 这个设置很少使用插件提供者指定的默认策略，有效值包括：read-only,read-write,nonstrict-read-write,transactional
- **hibernate.cache.use_structured_entries**-如果为 true，强制 Hibernate 按一种更人性化的方式在二级缓冲中存储数据。如果你想要直接在二级缓冲中“浏览”数据，可以使用这个设置，但是对性能有影响。
- **hibernate.cache.auto_evict_collection_cache**-启用或禁用自动清除双向关联中的集合缓冲，这种操作通常用于当关联由一方更改时。默认为禁用，因为跟踪这些状态是很影响性能的。然而如果你的应用程序不想在双向关联的两个方向上管理数据，特别是集合在一方已经缓冲了数据，或是集合一方有脏数据存在时 ，你可以启用这个选项。

## 11.2. 管理缓冲数据

在运行时 Hibernate 响应 Session 的操作，从二级缓冲中移入或移出数据。org.hibernate.Cache 接口(如果在 JPA 中就是 javax.persistence.Cache 接口)允许从二级缓冲中清理数据。

# 第十二章 12. 拦截器和事件

**目录**



如果应用程序能够影响 Hibernate 内部一些事件的发生，这是非常有用的。这将允许实现一些通用功能或扩展 Hibernate 的功能。

## 12.1. 拦截器 (Interceptors)

org.hibernate.Interceptor 接口提供从会话到应用程序的回调，通过这个回调，允许应用程序在保存、更新、删除、加载之前检查或操作持久化对象的属性。用途之一就是跟踪验证信息。例如，下面的示例展示一个拦截器实现类设置 createTimestamp 属性，就是当 Auditable（可校验）实体建立或更新时，建立或更新 lastUpdateTimestamp（最后修改的时间戳）属性。

注意

你即可以直接实现 Interceptor 接口也可以继承

org.hibernate.EmptyInterceptor。

一个拦截器既可以是 Session-scoped（会话的作用域）也可以 SessionFactory-scoped（整个会话工厂的作用域）

一个 Session-scoped 拦截器当会话打开时被定义。

```
Session session = sf.withOptions().interceptor( new
AuditInterceptor() ).openSession();
```

一个 SessionFactory-scoped 拦截器是在建立 SessionFactory 之前预先注册好的一个 Configuration（配置对象。除非打开一个 session 时显式指定拦截器，否则由 SessionFactory 打开的所有会话都将使用 SessionFactory-scoped 拦截器。SessionFactory-scoped 拦截器一这要设计成线程安全的。确保它是没有

保存具体 session 的状态，因为在一个可能的并发环境中，多个会话将使用这个拦截器。

```
SessionFactory sessionFactory = new Configuration()
                        .setInterceptor( new AuditInterceptor() )
                        ...
                        .buildSessionFactory();
```

## 12.2. Native（原生、本地）事件系统

如果你不得不对持久层的事件作出反应，你也可以使用 Hibernate *event（事件）* 体系。这个事件系统可以用来代替或补充现有的拦截器。

Session 接口的很多方法关联到事件类型。定义事件类型的一系列方法被声明为 org.hibernate.event.spi.EventType 枚举类型。这些方法生成的请求，会话产生了一个合适事件，并且将这个事件传递给配置好的事件监听器。应用程序可以自由的自定义监听器实现类（如 LoadEvent 事件的监听器，实现了 LoadEventListener 接口）。在这种情况下，它们的实现负类责处理 Session 请求的加载过程。

注意

参考这里，有关自定义监听器的信息。

监听器应该设计为无状态的；它们在请求之间共享，并且必须不保存任何实例属性值。

自定义监听器实现一个合适的接口，这个接口要与监听的事件相符合，也可以继承一个基类（其实，Hibernate 默认的监听器也继承自一个基类，这个基类为支持这种特点被设计成 non-final）。这里有一个自定义加载事件监听器的示例：

**示例 12.1. 自定义加载监听器**

```
public class LoadListenerExample implements LoadEventListener {
                // 这是 LoadEventListener 接口定义的唯一方法
                public void onLoad(LoadEvent event,
LoadEventListener.LoadType loadType)
                                throws HibernateException {
```

```
                        if
( !MySecurity.isAuthorized( event.getEntityClassName(),
event.getEntityId() ) ) {
                                throw
MySecurityException("Unauthorized access");
                        }
                }
        }
```

### 12.2.1. Hibernate 声明式安全

通常，session facade（门面模式）层中管理 Hibernate 应用程序中的声明式的安全。Hibernate 确保这些行为通过 JACC 认证，通过 JAAS 授权。这是可选功能，这个功能被放置在事件架构的顶层。
首先，你必须设置适当的事件监听才可以使用 JACC 授权。参考这里了解细节。下面示例中 org.hibernate.integrator.spi.Integrator 的实现类，支持这种特性。

**示例 12.2. 注册 JACC 监听器**

```
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_ENA
BLED );
                        if ( !isSecurityEnabled ) {
                                return;
                        }

                        final EventListenerRegistry
eventListenerRegistry =
serviceRegistry.getService( EventListenerRegistry.class );

        eventListenerRegistry.addDuplicationStrategy( JACC_DUPLICATIO
N_STRATEGY );

                        final String jaccContextId =
configuration.getProperty( Environment.JACC_CONTEXTID );

        eventListenerRegistry.prependListeners( EventType.PRE_DELETE,
new JACCPreDeleteEventListener(jaccContextId) );

        eventListenerRegistry.prependListeners( EventType.PRE_INSERT,
new JACCPreInsertEventListener(jaccContextId) );
```

```
        eventListenerRegistry.prependListeners( EventType.PRE_UPDATE,
new JACCPreUpdateEventListener(jaccContextId) );

        eventListenerRegistry.prependListeners( EventType.PRE_LOAD,
new JACCPreLoadEventListener(jaccContextId) );
                }
        }
```

你还要确定如何配置你的 JACC 提供者。参考你的 JACC 提供者文档。

## 12.3. JPA 回调

通过注释，JPA 还定义更多 callback（回调）的设定。

**表 12.1. 回调注释**

| 类型 | 描述 |
| --- | --- |
| @PrePersist | 在实体持久操作之前执行，实际是执行或者串联到其它方法（译者注：如果就一个这类回调方法就是执行方法，如果有多个这类回调就是口中联这些方法）。这个调用与持久化操作同步。 |
| @PreRemove | 在实体删除操作之前执行，实际是执行或者串联到其它方法（译者注：如果就一个这类回调方法就是执行方法，如果有多个这类回调就是口中联这些方法）。这个调用与删除操作同步。 |
| @PostPersist | 在实体持久操作之后执行，实际是执行或者串联到其它方法（译者注：如果就一个这类回调方法就是执行方法，如果有多个这类回调就是口中联这些方法）。这个调用与持久化操作同步。 这个调用被执行插入操作后。 |
| @PostRemove | 在实体删除操作之后执行，实际是执行或者串联到其它方法（译者注：如果就一个这类回调方法就是执行方法，如果有多个这类回调就是口中联这些方法）。这个调用与删除操作同步。 |
| @PreUpdate | 在实体更新操作之前执行。 |

| | |
|---|---|
| @PostUpdate | 在实体更新操作之后执行。 |
| @PostLoad | 实体已经加载入当前持久化上下文后执行，或实体被刷新后。 |

这里有两种可用的方法定义指定的回调处理：
第一个方法是注释实体自身的方法用于接收指定实体生命周期事件的通知。
第二个使用一个独立的实体监听器类。一个实体监听器是一个无状态的类，同时有一个无参构造。回调注释安置在这个类的方法上，而不是实体类上。然后实体类上用 javax.persistence.EntityListeners 注释与实体类监听器关联起来。

**示例 12.3. 指定 JPA 回调示例**

```
@Entity
        @EntityListeners( LastUpdateListener.class )
        public class Cat {
                @Id private Integer id;
                private String name;
                private Calendar dateOfBirth;
                @Transient private int age;
                private Date lastUpdate;
                //getters and setters

                /**
                 * Set my transient property at load time based on a
calculation,
                 * note that a native Hibernate formula mapping is
better for this purpose.
                 */
                @PostLoad
                public void calculateAge() {
                        Calendar birth = new GregorianCalendar();
                        birth.setTime(dateOfBirth);
                        Calendar now = new GregorianCalendar();
                        now.setTime( new Date() );
                        int adjust = 0;
                        if ( now.get(Calendar.DAY_OF_YEAR) -
birth.get(Calendar.DAY_OF_YEAR) &lt; 0) {
                                adjust = -1;
                        }
```

```
                        age = now.get(Calendar.YEAR) -
birth.get(Calendar.YEAR) + adjust;
                }
        }

        public class LastUpdateListener {
                /**
                 * automatic property set before any database
persistence
                 */
                @PreUpdate
                @PrePersist
                public void setLastUpdate(Cat o) {
                        o.setLastUpdate( new Date() );
                }
        }
```

这些方法可以混合使用，意味着你可以同时使用两种方法。
不管是实体类的回调方法还是实体监听器，它们签名的方法一定是 void 返回值。方法名不重要，因为这个方法这只是一个标记回调注释的标记位置而已。实体类回调方法的情况下，这个方法一定有一个无参的签名。回调实体监听类的回调方法一定是一个有一个参数的方法。这个参数既可以是 java.lang.Object（方便附加多个实体）也可以是一个指定的类型。
回调方法能够抛出 RuntimeException 异常，如果方法抛出 RuntimeException 异常，这个当前的事务，如果有，必须 rolled back(回滚)。
回调方法一定不能调用 EntityManager 或者 Query 的方法。
多个回调方法定义在一个生命周期事件中是可能的。在这种情况下，执行的顺序是按 JPA 规范进行的（具体参考 3.5.4 小节）：
相关的默认监听器首先被执行，按指定的 XML 文件中的顺序。参考 javax.persistence.ExcludeDefaultListeners 注释。
其次，与实体层级相关的实体监听器被执行，执行顺序按 EntityListeners 中定义的顺序。如果有多个实体层级相关的实体监听器，基类的监听器比子类的监听器先执行。参考 javax.persistence.ExcludeSuperclassListeners 注释。
最后，实体类的回调方法被执行。如果回调类型同时注释在实体类的方法中，与其子类的方法中，同时子类的方法没有重写。那么这些方法都会被调用，但是子类方法先调用。实体类也允许子类重写回调方法，在这种情况下，父类的方法不执行。如果父类中被重写的方法有回调注释，那么它也将执行。

# 第 13 章 HQL 与 JPQL

**概要**

讨论 HQL 与 JPQL 的语法与执行

**目录**

**相关主题**

原文
第 9 章抓取
第 4 章持久化上下文

Hibernate 查询语言（HQL）和 Java 持久化查询语言（JPQL）都是关注于对象模式的查询语言，与自然的 SQL 相似。JPQL 是 HQL 的子集（JPQL 的灵感来于 HQL）。一个 JPQL 一定是一个 HQL，相反就不正确了。
HQL 与 JPQL 都是以非类型安全（non-type-safe）的方式来执行查询操作。Criteria 查询提供类型安全的方式处理查询。参考第 14 章 *Criteria* 了解详细内容。

## 13.1. 大小写敏感性

除了 Java 的类名与属性名是区分大小写的，查询中的其它内容不区分大小写。因此 SeLeCT、sELEct、 SELECT 是一个意思，但是 org.hibernate.eg.FOO 不同于 org.hibernate.eg.Foo，就像 foo.barSet 不同于 foo.BARSET 一样。
注意
此文档在示例中使用小写关键字约定。

## 13.2. 语句（Stataement）类型

HQL 与 JPQL 都允许 SELECT,UPDATE 与 DELETE 语句执行。HQL 额外还允许 INSERT 语句，形式上与 SQL 的 INSERT-SELECT 相似。

处理批量更新与删除操作时要十分谨慎，因为这些操作可能导致数据库中的数据与持久化上下文的实体不一致。 通常情况下，批量更新与删除操作应该

运行一个新的持久化环境的事务中，或者在可能影响数据状态的操作（抓取操作，访问操作）之前进行。

--JPA 2.0 规范的 4.10 节

### 13.2.1. Select 语句

HQL 中 SELECT 的 BNF 表达式是：

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

最简单的 HQLSELECT 语句的形式如下：

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

除了 JPQL 需要 select_clause 从句之外，JPQL 的 select 语句与 HQL 的 select 语句完全相同，即使 HQL 不一定要 select_clause 从句，但是包括它们通常是一种好的做法。因为对于一个简单查询意图要明确，select_clause 从句的结果是很容易推断出来的。但是对于复杂的查询不总是这样。实际上，当解析 JPQL 查询时，Hibernate 并没有执行当前的 select_clause 从句，当你的应用是在可移植的 JPA 环境下工作时，你应该注意这些问题。

### 13.2.2. Update 语句

HQL 与 JPQL 的 UPDATE 语句 BNF 表达式是一样的：

```
update_statement ::= update_clause [where_clause]

        update_clause ::= UPDATE entity_name [[AS]
identification_variable]
                        SET update_item {, update_item}*

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

        new_value ::= scalar_expression |
```

```
                                        simple_entity_expression |
                                        NULL
```

UPDATE 语句，默认，不改变它们操作对象的 version 或者 timestamp 属性值。但是你可以强制 Hibernate 设置 version 或者 timestamp 的属性值，这可以用 versioned update 完成。它的实现方法是在 UPDATE 关键字后面加上 VERSIONED 关键字。注意，这是 Hibernate 特有的特性，不能用于可移植的方案中。自定义版本类型 org.hibernate.usertype.UserVersionType，不能与 update versioned 语句连用。

UPDATE 的执行是通过 org.hibernate.Query 或者 javax.persistence.Query 的 executeUpdate 方法。这个方法的命名是源自于 JDBC 的 java.sql.PreparedStatement 中的 executeUpdate，以使大家不会显得陌生。executeUpdate()方法的返回值是一个 int，代表操作影响的对象数量。这个数量可能是影响的行数，也可能不是。HQL 批量操作的结果可能是实际执行了多条 SQL 语句（如 连接了子类）。这个返回值代表实际影响的实体数量。使用 JOINED 继承的层级结构中，一个子类的删除可能实际删除了不只是映射中指定的子类，因为子类也可能是有关联的。

**示例 13.1. UPDATE 语句**

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

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

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

重要的

无论是 UPDATE 还是 DELETE 语句，不允许用于隐式 join 的结果。现在它们已经也不允许用于显式 join 的结果。（译者注：原文就是这样写的。）

### 13.2.3. Delete 语句

HQL 与 JPQL 的 DELETE 语句 BNF 表达式是一样的：

```
delete_statement ::= delete_clause [where_clause]

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

DELETE 同样也使用 org.hibernate.Query 或者 javax.persistence.Query 的 executeUpdate 方法完成。

### 13.2.4. Insert 语句

HQL 增加了定义 INSERT 语句的能力， 但是 JPQL 没有这样的能力。HQL INSERT 语句的 BNF 表达式：

```
insert_statement ::= insert_clause select_statement

        insert_clause ::= INSERT INTO entity_name (attribute_list)

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

attribute_list 与 SQLINSERT 语句的 column specification（列规范）相似。对于存在映射继承的实体，只有直接定义在实体上的属性可以被

attribute_list 使用。父类属性不能使用子类属性，这是没有意义。换句话说，INSERT 语句本质上不支持多态。

select_statement 可以是任何有效的 HQL select 查询。但是注意，返回的类型一定与 insert 期望的类型相匹配。目前，这个检查在查询编译时进行，而不是在数据库中进行。这会引起很多的问题，当 Hibernate 类型转换到相似的类型而不是相同的类型时。例如，一个映射定义为 org.hibernate.type.DateType，与之相对应的属性定义为 org.hibernate.type.TimestampType，这可能会引起匹配错误，即使数据库可以区分这种不同或可以处理这种转换。也同样会出现错误。

对于 ID 属性，在 insert 语句中提供两种可选方案。第一种：你可以在 attribute_list 中显式指定 ID 属性，在这种情况下，ID 值从适当的 select 表达式中得到；第二种：你可以在 attribute_list 中忽略 ID 值，在这种情况下，会使用一个生成的值。第二种方案只能用于使用数据库自身的 ID 生成器时，假设在第二种这方案时使用内存的 ID 生成器，将在解析过程中抛出异常。

对于乐观锁锁定的属性，在 insert 语句中也提供两种可选方案。第一种：你可以在 attribute_list 中指定属性，在这种情况下，属性值从相应的 select 表达式中得到；第二种：你可以在 attribute_list 中忽略属性值，在这种情况下，使用 org.hibernate.type.VersionType 定义相应的 seed value（种子值）。

**示例 13.2. INSERT 语句**

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

## 13.3. FROM 子句

FROM 子句负责定义查询的其他部分可以使用的对象模型的范围。它也负责定义所有的“identification variables（标识变量）（译者注：别名）”可用于查询的其它部分。

### 13.3.1. 标识变量

标识变量通常被称为别名。FROM 子句中的引用对象类与标识变量相关联，然后标识变量可以被查询的其它部分使用。
大多数情况下，声明标识变量是可选的，但是声明它们是很好的做法。
标识变量一定要遵循 JAVA 有效的标识规范。
根据 JPQL，标识变量必须看成忽略大小写。好的做法是，在整个查询中，你应该使用相同的策略来提供标识变量。换句话说，JPQL 不要求大小写，因为 Hibernate 可以正确处理它们，但是最好不要这样做（译者注：建议区分大小写，但是要注意本质上是不区分的）。

### 13.3.2. Root(根)实体引用

根实体引用，或者说是 JPA 称为 range variable declaration（范围变量声明），是应用具体引用的映射实体类型。它们不是组件、插件类型名。联合、集合有不同的处理方法以后讨论。
根实体引用的 BNF 表达式：

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

**示例 13.3. 简单的查询**

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

我们看到这个查询声明了根实体引用 com.acme.Cat 对象模型类型。此外，它还声明了一个别名 c；这就是标识变量。
通常根实体引用只命名为 entity name（实体的短名），而不用实体的 FQN(译者注：fully qualified name 全名)。默认实体命名为 unqualified（不标准）（译者注：无包名）的类名，这里的 Cat。

**示例 13.4. 根实体引用使用实体名的简单查询**

```
select c from Cat c
```

也可以指定多个根实体引用。甚至同一个实体命名为不同的名称。

**示例 13.5. 多根实体引用简单示例**

```
// build a product between customers and active mailing campaigns so
we can spam!
        select distinct cust, camp
        from Customer cust, Campaign camp
        where camp.type = 'mail'
          and current_timestamp() between camp.activeRange.start and
camp.activeRange.end
// retrieve all customers with headquarters in the same state as
Acme's headquarters
        select distinct c1
        from Customer c1, Customer c2
        where c1.address.state = c2.address.state
          and c2.name = 'Acme'
```

### 13.3.3. 显式 join

FROM 子句也可以使用 join 关键字显式 join。这里 join 可以是 inner 或者 left outer 的 join。

**示例 13.6. 显式 inner join**

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

        // 相同的查询显示使用'inner'
        select c
        from Customer c
                inner join c.chiefExecutive ceo
        where ceo.age < 25
```

**示例 13.7. 显式 left (outer) join**

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

        // 功能与上面的一样，只是使用 left outer
        //functionally the same query but using the
        // 'left outer' phrase
        select distinct c
        from Customer c
                left outer join c.orders o
        where o.value > 5000.00
          or c.status = 'preferred'
```

一个显式 join 重要的用例是定义 FETCH JOINS，这个可以替代懒加载。参考一个示例，Customer（客户类）类与之关联的 orders（定单类）集合。

**示例 13.8. join Fetch**

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

正如你从示例中看到的，一个 fetch join 是通过在 join 后加上 fetch 完成的。在示例中，我们使用 left outer join，是因为我们想要返回所有用户的订单（包括没有订单的用户）。inner join 也可以用于抓取。但是 inner join 有过滤功能。在示例中，使用 inner join 代替将会在结果集中过滤掉没有定单的用户。

重要的

Fetch joins 在子查询中无效。

要注意，当与抓取关联的集合有进一步限制时；获得的集合也同样会有限制。出于此原因 ，最好的方法是不对这样的抓取设定标识变量。但是嵌套的抓取除外。

Fetch join 不能用于分页查询（即，setFirstResult/ setMaxResults）。同样也不能用于带有 scroll（滚动）或者 iterate（迭代）特性的 HQL 中。

HQL 也定义了 WITH 子句来限定 join 的条件。再次提醒这是 HQL 的使用的特性，不能用于 JPQL。

**示例 13.9. 带有 with 子句的 join**

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

这里有一个重要发的区别：在生成的 SQL 语句中。由 HQL with clause（with 子句）生成的是 SQL on clause（on 条件）的一部分（译者注：不是 where 从句，网上很多报错都是没明白这点），与之相对的是其它查询中，HQL/JPQL 的 where 条件生成 SQL 中 where clause（where 子句）的一部分。这个区别在此示例中不是很重要，但是在一些复杂查询中，with clause 有时是必须的（译者注：比如有聚合函数时）。

显式的 join 可以引用 关联、组件、插件的属性。关联集合引用的更深内容参考 13.3.5, “集合成员引用”。当用于组件、插件的属性时，这个显式 join 就是一个简单的逻辑 join，并不是 SQL 语句的物理 join。

### 13.3.4. 隐式 join（path 表达式）

另一种将对象加入查询范围中的方法是通过隐式 join 或 path 表达式。

**示例 13.10. 简单的隐式 join**

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

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

一个隐式 join 总是由 identification variable（标识变量、别名）开始，后面跟一个导航运算符（.），后面是别名代表的对象的属性。在本例中，identification variable（标识变量、别名）是 c，它代表 Customer 实体。c.chiefExecutive 属性代表 Customer 的 chiefExecutive 属性。chiefExecutive 属性是一个关联的引用，再进一步导航到 age 属性。

重要的

如果属性是实体关联（非集合）或者是组件、插件的，可以进一步导航。基本数据类型与集合的关联不能进一步导航。

如示例所示，隐式 join 出现在 FROM clause（from 子句）之外，但是可以影响 FROM clause（from 子句）。隐式 join 总是被当成 inner join 来处理。多个引用相同的隐式 join，总是指向同一个逻辑与 SQL 语句的物理 join（译者注：不会多次 join）。

**示例 13.11. 重用隐式 join**

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

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

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

与显式 join 一样，隐式的 join 可以引用关联或者 component/embedded（组件、插件）的属性。关联集合引用的更深内容参考 13.3.5, "集合成员引用"。当用于 component/embedded（组件、插件）的属性时，这个显式 join 就是一个简单的逻辑 join，并不是 SQL 语句的物理 join。与显式 join 不同的是，隐式 join 可以引用基本的数据类型字段，只要 path 表达式在字段上结束就可以（译者注：不能再. 下去了）。

### 13.3.5. 集合成员引用

引用关联的 collection-valued 实际上是针对集合 *values（值）* 的引用

**示例 13.12. 集合引用**

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

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

在示例中，标识变量（别名）o 实际上引用对象模型 Order 类型，就是 Customer#orders 关联。
这个示例中也展示了 join 的替换语法 IN。两种语法是等价的， 应用中选择哪个是个人喜好的问题。

#### 13.3.5.1. 特殊情况--条件路径表达式

我们已经说过了关联的 collection-valued 实际上是集合的 *values（值）*，基于集合类型的引用，也有一套显示条件表达式。

**示例 13.13. 带有条件的集合引用**

```
// Product.images is a Map<String,String> : key = a name, value =
file path
        //Product.images 是一个 Map<String,String>: key = 名称, value
= 文件路径
        // select 产品 123 的所有图片路径（map 的 value）
        select i
        from Product p
                join p.images i
        where p.id = 123

        // 与上面一样
        select value(i)
        from Product p
                join p.images i
        where p.id = 123


        // 查询所有编号为 123 的产品图片的 KEY
        select key(i)
        from Product p
                join p.images i
        where p.id = 123


        //查询所有编号为 123 的产品图片 map 实体
        select entry(i)
        from Product p
```

```
        join p.images i
where p.id = 123


//客户 123 的所有定单项目的总合
select sum( li.amount )
from Customer c
               join c.orders o
               join o.lineItems li
where c.id = 123
  and index(li) = 1
```

VALUE

指集合的值，等价与没有限制条件。用于显式展示集合对象。适用于任何类型的集合引用。

INDEX

按 HQL 规则，可以用于 Map 与 List，代表指定的 javax.persistence.OrderColumn 注释的值，对于 Map 是指 Key，对于 List 是指下标（即 OrderColumn 的值）。然而 JPQL 中，使用 List 时是一样的，但是使用 Map 时，增加了一个 KEY 值。在可开发可移植应用时，要注意这些区别。

KEY

只能用于 Map，是指 Map 的 Key。如果 Key 是一个对象，可能导航到更深一层。

ENTRY

只能用于 Map，是指 Map 的原型 java.util.Map.Entry（Key 与 Value 的组合）。ENTRY 只能做为最终的路径（译者注：不能向下导航了，也就是.了），只能用于 Select 子句。

参考 13.4.9,"集合相关的表达式"了解更多细节 。

### 13.3.6. Polymorphism（多态）

HQL and JPQL 查询天生支持多态

```
select p from Payment p
```

查询显式指定 Payment。然而所有 Payment 的子类也是有效的查询。所以 Payment 以及其的子类 CreditCardPayment 与 WireTransferPayment 三种类型都是有效的查询。并且查询可以返回所有三种类型的对象实例。

极端逻辑

在 HQL 中 from java.lang.Object 是完全有效的！它将返回应用程序中定义过的所有类型。

这种多态是可以改变的，通过 org.hibernate.annotations.Polymorphism 注释（全局时可以通过 Hibernate-specific），或者在查询中通过实体类型表达式限制。

## 13.4. 表达式

从本质上来说表达式引用的是基本类型或者原型值。

### 13.4.1. 标识变量

参考 13.3, "FROM 子句".

### 13.4.2. 路径表达式

再次参考 13.3, "FROM 子句".

### 13.4.3. 文本说明

字符串用一对单引号，如果字符串中有单引，使用两个单引号。

**示例 13.14. 字符串示例**

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

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

数字允许用以下几种不同的形式。

**示例 13.15. 数字示例**

```
// 简单的整型
select o
from Order o
where o.referenceNumber = 123

// 简单的长整
select o
from Order o
where o.referenceNumber = 123L

// 小数标记（double）
select o
from Order o
where o.total > 5000.00

// float 标记
select o
from Order o
where o.total > 5000.00F

// 科学计数法标记
select o
from Order o
where o.total > 5e+3

// float 科学计数法标记
select o
from Order o
where o.total > 5e+3F
```

在科学计数法中，E 不区分大小写。

具体数字类型与 JAVA 表示数据的后缀相同。因此 L=long, D=double, F=float。

实际使用时不区分大小写。

布尔值为 TRUE 与 FALSE，也不区分大小写。

枚举也可以在文本中使用。但是一定使用全限定名称（FQN）。HQL 也可以用相同的方式处理常量，但是 JPQL 不支持常量。
实体名也可以在文本中使用。参考 13.4.10, “实体类型”
时间/日期通过 JDBC 转义语法使用：{d '2013-03-27'} 日期，{t '10:43:59'} 时间，{ts '2013-03-27 10:43:59.616'}（millis 可选）时间戳。这些需要你的 JDBC 驱动支持。

### 13.4.4. 参数

HQL 支持以下的 3 种形式。JPQL 不支持 HQL 风格的位置参数。最好不要在一个查询中混合使用多个参数形式。

#### 13.4.4.1. 命名参数

命名参数使用一个冒号后面跟一个标识的方式声明--:aNamedParameter。一个查询中可以多次出现相同的命名参数。

**示例 13.16. 命名参数**

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

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

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

#### 13.4.4.2. JPQL 位置参数

JPQL 的位置参数一个问号跟一个序号的方式声明--?1,?2。序号从 1 开始，与命名参数一样，在一个查询中可能多次出现。

**示例 13.17. JPQL 位置参数**

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

        // HQL - 正如你所见，与命名参数相似
        //      in terms of API
        List customers = session.createQuery( queryString )
                        .setParameter( "1", theNameOfInterest )
                        .list();

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

#### 13.4.4.3. HQL 位置参数

HQL 风格的位置参数与 JDBC 位置参数语法相似。使用一个?号声明，后面没有序号。这就造成如果两个参数值相同，你也要绑定 2 次。
这种形式应该废除，可以不久的将来就不用了。

### 13.4.5. 算术运算

算术运算也是有效的表达式。

**示例 13.18. 数值算术运算示例**

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

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

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

算术运算符的获取结果时，使用以下规则：
如果两边操作数为 Double/double，结果为 Double；
如果两边操作数为 Float/float，结果为 Float；
如果两边操作数为 BigDecimal，结果为 BigDecimal；
如果两边操作数为 BigInteger，结果为 BigInteger，（除了除法，结果没有进一步定义）；
如果两边操作数为 Long/long，结果为 Long，（除了除法，结果没有进一步定义）；
其它（假定两边操作数为 Integer），结果为 Integer，（除了除法，结果没有进一步定义）；
时间计算也是支持的，虽然有很多限制。这些限制一部分来源于是数据库差异，一部分来源于查询语言本身的差异，比如缺乏统一的 INTERVAL 定义。

### 13.4.6. Concatenation（串联）（运算）

HQL 定义了串联运算符以增加了串联运算（CONCAT 函数）。JPQL 中没有定义，因此可移植应用要回避它。串联操作符与 Sql 的连接运算符相同-- ||

**示例 13.19. 串联运算**

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

参考 13.4.8, "Scalar 函数"，了解 concat()函数的细节。

### 13.4.7. 聚合函数

聚合函数在 HQL 与 JPQL 中都是有效的表达式。语义上与 SQL 相同，所支持的聚合函数有：
COUNT(包括：不同/所有)--返回结果类型永远是 Long。
AVG--返回值计算结果的平均值。返回结果类型永远是 Double。
MIN--返回结果类型与参数类型一样。
MAX--返回结果类型与参数类型一样。
SUM--对于整数值（除了 BigInteger），结果为 Long。对于浮点值（除了 BigDecimal），结果类型为 Double。对于 BigInteger 值，结果类型为 BigInteger。对于 BigDecimal 值，结果类型为 BigDecimal。

**示例 13.20. 聚合函数**

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

        select count( distinct c.name )
        from Customer c

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

聚合经常与分组一起。有关分组的信息参考 13.8, "分组"

### 13.4.8. Scalar（标量）函数

HQL 与 JPQL 定义一些标准函数，这些函数可以不管底层数据库直接使用。HQL 还可以理解通过 Dialect 与应用程序定义的附加函数。

#### 13.4.8.1. 标准函数-JPQL

下面这些函数是 JPQL 支持的函数列表。应用程序为了可移植性，应当支持这些函数。

CONCAT

字符串连接函数。两个或多个字符串参数被连接在一起。

SUBSTRING

提取字符串的一部分值。

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

第二个参数是开始位，第三个参数（可选）是提取长度

UPPER

指定的字符串变成大写。

LOWER

指定的字符串变成小写。

TRIM

与 SQL 的 trim 函数功能一样。

LENGTH

返回字符串的长度

LOCATE

字符串是否包含另一个字符串。

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

第三个参数（可选）标注开始查询的位置。

ABS

计算一个数字的绝对值。

MOD

计算第一个参数除第二个参数的余数。

SQRT

计算数字的平方根。

CURRENT_DATE

返回当前日期。

CURRENT_TIME

返回当前的时间。

CURRENT_TIMESTAMP

返回当前的时间戳。

#### 13.4.8.2. 标准函数--HQL

超出了 JPQL 标准函数的范围，HQL 多出一了一些额外的函数，可以不考虑底层数据库，直接使用。
BIT_LENGTH

返回二进抽数据长度。
CAST

执行 SQL 的 cast,转换的目标是 Hiberante 映射的类型。参考数据类型的相关章节了解更多的信息。
EXTRACT

执行 SQL 提取时间日期值。提取是指提取时期、时间的分（年，时间）。参考下面的缩写形式。
SECOND

秒的缩写。
MINUTE

分钟缩写。
HOUR

小时缩写。
DAY

日期缩写。
MONTH

月缩写。
YEAR

年缩写 。
STR

转换为字符形式的日期。

#### 13.4.8.3. 非标准函数

Hibernate 方言可以为指定的数据库产品注册额外的可用函数。这些函数在 HQL 可用（在 JPQL 中，只有明确指定 Hibernate 做为 JPA 提供者时可用）。然而，

它们只在特定的数据库/方言中可用。如果你的应用将用于可移植的平台，应该避免使用这类函数。
应用程序开发者可以使用自己的函数集。这通常是指自定义的 SQL 函数或 SQL 代码段的别名。这些函数是通过 org.hibernate.cfg.Configuration 类的 addSqlFunction 方法声明的。

### 13.4.9. 集合相关的表达式

这是一专用于集合值相关的表达式。通常它们是缩写形式已经说明了它们的目的。
SIZE

计算集合的大小，等价于子查询。
MAXELEMENT

只能用于基本数据类型的集合。是指最大值，内部使用 SQL 聚合函数 max 确定返回值。
MAXINDEX

用于索引集合。是指最大的索引值（关键字/下标），内部使用 SQL 聚合函数 max 确定返回值。
MINELEMENT

只能用于基本数据类型的集合。是指最小值，内部使用 SQL 聚合函数 min 确定返回值。
MININDEX

用于索引集合。是指最小的索引值（关键字/下标），内部使用 SQL 聚合函数 min 确定返回值。
ELEMENTS

是指将集合的元素做为一个整体使用。只能出现的 where 从句中。经常与 ALL，ANY ，SOME 连用。
INDICES

类似 elements，不过是指 indices 将集合的索引（关键字/下标位）做为一个整体使用。

**示例 13.21. 集合相关的表达式**

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

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

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

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

        // 以标准的 jpql 方式重写上面的查询 :
        select m
        from Cat as m, Cat as kit
        where kit member of m.kittens

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

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

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

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

有序的集合元素（arrays, lists, maps）才能指定索引操作符。

**示例 13.22. 索引操作符**

```
select o
```

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

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

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

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

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

也可以参考 13.3.5.1, "特殊情况--条件路径表达式"，因为这是一种很好的替代方案。

### 13.4.10. 实体类型

我们也可以将这实体的类型称作表达式。这在实体继承层级结构中很有用。类型可以使用 TYPE 函数表示，函数中可以使用类型的别名代表实体。实体的名称是引用实体类型的一种方式。另外实体类型可以参数化，在这种情况下，实体的 Java 类引用将被绑定为参数值。

**示例 13.23. 实体类型表达式**

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

        select p
```

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

HQL 还有一种过时的形式针对实体类型，如果你不喜欢 TYPE 这种形式可以考虑老的方式。老方式是用 p.class 而不是用 type(p)。这里提一下是为了完整性。

### 13.4.11. CASE 表达式

支持简单形式或者搜索形式，以及 2 种 SQL 定义的缩写形式（NULLIF 与 COALESCE）。

#### 13.4.11.1. 简单 CASE 表达式

简单形式使用下面的语法：

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

**示例 13.24. 简单 CASE 表达式**

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

        //  对于这种 null 值检查，大多数数据库定义了另外一种缩写形式，
如：

        select nvl( c.nickName, '<no nick name>' )
        from Customer c

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

        // 标准的 coalesce 缩写可以达到同样的效果：

        select coalesce( c.nickName, '<no nick name>' )
```

```
        from Customer c
```

#### 13.4.11.2. 搜索 CASE 表达式

搜索形式使用下面的语法

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

**示例 13.25. 搜索 CASE 表达式　Searched case expression example**

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

        // 更简洁的缩写
        select coalesce( c.name.first, c.nickName, '<no first name>' )
        from Customer c
```

#### 13.4.11.3. NULLIF 表达式

NULLIF 是一种缩写的 CASE 表达式，是指两个操作值相等时返回为空。

**示例 13.26. NULLIF 示例**

```
        //返回改名的客户
        select nullif( c.previousName.last, c.name.last )
        from Customer c

        //等价的 CASE 表达式
        select case when c.previousName.last = c.name.last then null
                            else c.previousName.last end
```

```
        from Customer c
```

#### 13.4.11.4. COALESCE 表达式

COALESCE 是 CASE 表达式的缩写。它返回第一个非空的操作数。我们上面已经看到过 COALESCE 的例子了。

## 13.5. SELECT 子句

SELECT 子句确定查询返回哪些对象或值。它的表达式参考 13.4, "表达式"，这些表达式都时可用的，除非特别声明不可用于 select。还可以参考 13.10, "查询 API"，那里讨论具体的 SELECT 子句依赖值类型的细节信息。
这里有一种特殊的表达式，这个表达式只在 select 从句中有效。Hibernate 称之为"dynamic instantiation（动态实例化）"。JPQL 也支持这种特性，称之为"constructor expression（构造表达式）"。

**示例 13.27. 动态实例化--构造**

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

查询返回的结被封装成 java 类型安全对象而不是作为 Object[]进行处理（参考 13.10, "查询 API"）。这些类的引用一定是全类名，并且类必须有相匹配的构造函数。
这些类是不需要映射的。如果它看做一个实体，那么结果返回的实例处于 NEW（新建）状态（不是已托管状态）。
这部分内容 JPQL 也支持，HQL 还支持额外的"dynamic instantiation（动态实例）"特性。
首先，查询返回的 scalar（标量）结果集可以是一个 List 而不是 Object[]：

**示例 13.28. 动态实例化--List**

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

查询的结果是 List<List> 而不是 List<Object[]>

HQL 也支持在 map 封装的 scalar（标量）结果集。

**示例 13.29. 动态实例化--map**

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

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

查询的结果集是 List<Map<String,Object>>而不是 List<Object[]>。map 中的 key 值是通过 select 表达式中的别名定义的。

## 13.6. Predicates（谓词）

谓词构成的基于是 where 从句，having 从句,搜索表达式。它们一般解析成 TRUE 或 FALSE，虽然 boolean 比较涉及到 null 值，但是它们通常解析为 UNKNOWN。

### 13.6.1. 关系比较

比较涉及到以下比较运算符之一：ors - =, >, >=, <, <=, <>]> 。HQL 也定义了!=，这与 <>是相同的。两边的操作数一定是相同的数据类型。

**示例 13.30. 关系比较**

```
//数字比较
select c
from Customer c
where c.chiefExecutive.age < 30

// 字符串比较
select c
from Customer c
where c.name = 'Acme'

//日期比较
select c
from Customer c
where c.inceptionDate < {d '2000-01-01'}

//枚举比较
select c
from Customer c
where c.chiefExecutive.gender = com.acme.Gender.MALE

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

//实体类型比较
select p
from Payment p
where type(p) = WireTransferPayment

//实体值比较
select c
from Customer c
where c.chiefExecutive = c.chiefTechnologist
```

比较也涉及到子查询的限定符--ALL, ANY,SOME，SOME 与 ANY 是相同的。
如果比较子查询结果的所有值为 true，ALL 限定符解析为 true。如果子查询的结果为空，ALL 限定符解析为 false。

**示例 13.31. ALL 子查询**

```
//查询每次得分至少为 3 分的玩家。（译者注：是每次而不是某次）
select p
from Player p
where 3 > all (
        select spg.points
        from StatsPerGame spg
        where spg.player = p
)
```

如何子查询的结果至少一个为 true，ANY/SOME 限定符解析为 true。如果子查询的结果为空，ANY/SOME 限定符解析为 false。

### 13.6.2. 空值谓词

用于检查值是否为空。可以应用于属性引用，实体引用，参数值。 HQL 额外允许应用于 component/embeddable(组件/插件)类型。

**示例 13.32. 检查空值**

```
//查询有地址的人
select p
from Person p
where p.address is not null

// 查询没有地址的人
select p
from Person p
where p.address is null
```

### 13.6.3. Like 谓词

执行字符串的相似比较。语法为：

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

```
[ESCAPE escape_character]
```

语义与 SQL 的 like 表达式相同。pattern_value 是指在 string_expression 中试图匹配的的带有通配符的模板。就如 SQL 的 like 一样。pattern_value 可以使用两种通配符："_"与"%"。"_"匹配任意单一字符。"%"匹配任意数量的多个字符。

可选的 escape_character 是定义转义字符的，用于指定 pattern_value 中的转义字符，此字符用于转义"_"与"%"。这通常用于搜索的模板包括"_"或"%"时使用。

**示例 13.33. Like 谓词**

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

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

       //发现以“sp_”开始的字符串 find any with name starting with
"sp_"
       select sp
       from StoredProcedureMetadata sp
       where sp.name like 'sp|_%' escape '|'
```

### 13.6.4. Between 谓词

类似 SQL 的 between 表达式。判断一个值是否在 2 个值之间的范围中。所有的操作值必须是可比较的。

**示例 13.34. Between 谓词**

```
select p
       from Customer c
```

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

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

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

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

### 13.6.5. In 谓词

IN 检查指定的值是否出现在列表中。语法如下：

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

        single_valued_list ::= constructor_expression |
                            (subquery) |
                            collection_valued_input_parameter

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

single_valued_expression 的类型与 single_valued_list 中每个值的类型要一致。JPQL 可用类型有：字符串，数字，日期，时间，时间戳，枚举。在 JPQL 中 single_valued_expression 的类型参考如下 ：

"state fields"，这是指简单的属性，具体说就是 不包括 关联属性与 component/embedded(组件/插件)属性。

实体表达式。参考 13.4.10, "实体类型"

HQL 的 single_valued_expression 提供更广泛的表达类型。允许单值的关联，component/embedded（组件/插件）属性都是允许的，虽然这些特性要依赖底层数据库的支持（主要是"row value constructor syntax（行值构造语法）"与 tuple 的定

义方式）。另外 HQL 不以任何方法限制值的类型，但是应用开发者应该意识到相同的数据类型可能在不同数据库提供商中有不同的限制。这也是 JPQL 加上限制的原因。

List 的值可以有多种来源。constructor_expression 与 collection_valued_input_parameter 中的 List 不能为空，至少要有一个值。

**示例 13.35. In 谓词**

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

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

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

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

        // JPQL 不兼容
        select c
        from Customer c
        where c.name in (
                ('John','Doe'),
                ('Jane','Doe')
        )

        // JPQL 不兼容
        select c
        from Customer c
        where c.chiefExecutive in (
                select p
```

```
                from Person p
                where ...
        )
```

### 13.6.6. Exists 谓词

Exists 表达式测试结果是否存在于子查询中。如果子查询中包含结果就返回 true,如果子查询为空，返回 false(译者注：原文中用的是“非真”，不知道是否有特殊的含意);

### 13.6.7. 空集合谓词

IS [NOT] EMPTY 表达式应用于 collection-valued 路径表达式。它检查特定集合中是否关联了值（译者就：空值检查）。

**示例 13.36. 空集合表达式**

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

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

### 13.6.8. 集合成员谓词

[NOT] MEMBER [OF]表达式应用于 collection-valued 路径表达式。它检查一个值是否是指定集合的成员。

**示例 13.37. 集合成员表达式**

```
select p
```

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

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

#### 13.6.9. NOT 谓词

NOT 操作符用于否定谓词后面的内容。如果后面是 true,那么 NOT 就解析为 false。如果后面是 false,那么 NOT 就解析为 true。如果后面是 unknown,那么 NOT 就解析为 unknown。

#### 13.6.10. AND 谓词

AND 用于连接两个表达式。如果两个表达式都是 true,结果为 true,只要有表达式为 unknown，结果为 unknown，其它情况下，都是 false。

#### 13.6.11. OR 谓词

OR 用于连接两个表达式。如果有一个表达式为 true，结果为 true。如果两个表达式都为 unknown，结果为 unknown（译者注：注意与 and 的区别，都是 unknown 才为 unknown）。其它都为 false。

### 13.7. WHERE 子句

WHERE 子句由各种谓词组成，它检查与谓词相匹配的潜在的行。因此，where 子句用于约束 Select 的返回结果，限制 update,delete 语句的作用范围。

### 13.8. 分组

GROUP BY 子句允许建立各种分组的聚合结果。考虑下面的例子：

**示例 13.38. Group-by（分组）示例**

```
//得到所有定单的总合
select sum( o.total )
from Order o

//按客户分组，得到每个客户的定单总合
select c.id, sum( o.total )
from Order o
       inner join o.customer c
group by c.id
```

第一个查询得到了全部定单的总合，第二个查询得到了每个客户订单的总合；是按客户分组后的结果。

在分组查询中，where 子句应用于没有聚合的值（实际上它决定是否行将进入到聚合中）。The HAVING 子句也限制了结果，但它操作的是聚合的数据。（译者注：having 中是可以加上聚合函数的，但是 where 不可以，where 是全部数据的筛选，having 是分组后的数据筛选）在示例 13.38, “Group-by （分组）” 中，我们得到了所有定单的总合，如果继承处理数据，我们只聚焦在客户定单总合大于$10,000.00 的数据：

**示例 13.39. Having 示例**

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

HAVING 从句与 where 从句使用相同的规则，也是由谓词构成。但是 HAVING 是在分组之后，聚合已经完成了（译者注：原文有误？应该是有多种情况，可能在计算聚合时进行，也可能是聚合之前进行），WHERE 是在分组之前进行。

## 13.9. 排序

查询的结果是可以排序的。ORDER BY 子句使用选定的值用于结果的排序。用于排序的有效选定值包括：

状态字段

component/embedded(组件/插件)属性

scalar expressions（标量表达式）比如算术操作符，函数等等。

select 子句中以上所有类型的标识变量声明。

另外，JPQL 认为 order-by 子句中的所有引用必须在 select 从句中出现。HQL 没有这样的限制，但是如果应用程序希望数据库可移植，那么你必须注意不是所有的数据库都支持 order-by 子句的引用没有被 select 从句引用。

order-by 子句中的每个表达式都可以按你希望的排序方向排序，可以是 ASC（升序）或者 DESC（降序）。空值可以安置在前面或后面，这要使用 NULLS FIRST（空值在前）或 NULLS LAST（空值在后）分别设置 。

**示例 13.40. Order-by 示例**

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

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

## 13.10. 查询 API

### 13.10.1. Hibernate 查询 API

在 Hibernate 中，HQL/JPQL 查询是从 Session 中获得的 org.hibernate.Query 对象。如果 HQL/JPQL 是命名查询，使用 Session#getNamedQuery；其它使用 Session#createQuery。

**示例 13.41. 获取查询引用--Hibernate**

```
Query query = session.getNamedQuery( "my-predefined-named-query" );
Query query = session.createQuery(
                "select e.id, e.name from MyEntity e"
        );
```

查询接口可以控制查询的执行。例如，我们可能希望指定执行超时或者控制缓冲操作。

**示例 13.42. 基本查询用法--Hibernate**

```
Query query = ...;
        // in seconds
        query.setTimeout( 2 );
        // write to L2 caches, but do not read from them
        query.setCacheMode( CacheMode.REFRESH );
        // assuming query cache was enabled for the SessionFactory
        query.setCacheable( true );
        // add a comment to the generated SQL if enabled with the SF
        query.setComment( "e pluribus unum" )
```

了解全部细节，参考查询的 java 文档。

重要的

查询的 hint(提示)就是数据库的 hint(提示)。它们根据 Dialect#getQueryHintString 直接添加到的生成的 SQL 语句中。另一方面，JPA 的查询 hint(提示)与 JPA 提供者（Hibernate）有关。因此虽然它们调用相同的内容，但是会有完全不同的作用。因此要注意 Hibernate 的查询

hint(提示)通常不可能跨数据库移植，除非在加入代码前首先检查一下 Dialect（方言）。

清理缓冲的详细介绍在这里。锁的详细介绍在这里。只读状态概念的介绍在第四章 *持久化上下文*

Hibernate 也允许应用程序参与建立查询结果的过程，它通过 org.hibernate.transform.ResultTransformer 规范实现。参考 java 文档以及 Hibernate 提供的实现了解更多细节。

我们执行查询前做的最后一件事是为查询中定义的参数绑定值。查询为了这个目的定义了很多重载的方法。这些方法的参数大多数情况下是值，也可以是 Hibernate 的类型。

**示例 13.43. 参数绑定--Hibernate**

```
Query query = session.createQuery(
                "select e from MyEntity e where e.name like :filter"
        );
        query.setParameter( "filter", "D%", StringType.INSTANCE );
```

Hibernate 一般可以理解上下文中查询传递过来的参数的目标类型。在上面的例子，因为我们使用了 LIKE 比较，后面参数一定是字符串类型，Hibernate 可以自动判断这个类型；所以上面的例子是可以简化的。

**示例 13.44. 参数绑定（判断类型）--Hibernate**

```
Query query = session.createQuery(
                "select e from MyEntity e where e.name like :filter"
        );
        query.setParameter( "filter", "D%" );
```

对于绑定的公共类型比如 string, boolean, integer，等等，可以使用简写形式。

**示例 13.45. 参数绑定（简写形式）--Hibernate**

```
Query query = session.createQuery(
                "select e from MyEntity e where e.name like :filter"
        );
        query.setString( "filter", "D%" );

        query = session.createQuery(
                "select e from MyEntity e where e.active = :active"
        );
        query.setBoolean( "active", true );
```

在执行方面，Hibernate 提供了 4 个不同的方法，2 个是常用的方法：

Query#list --执行 select 查询，并返回一个结果列表。

Query#uniqueResult--执行 Select 查询，并返回一个单一结果，如果有多个结果值，就抛出异常。

**示例 13.46. list()与 uniqueResult()方法**

```
List results =
                session.createQuery( "select e from MyEntity e" )
                        .list();

String qry = "select e from MyEntity e " +
                        " where e.code = :code"
        MyEntity result = (MyEntity) session.createQuery( qry )
                .setParameter( "code", 123 )
                .uniqueResult();
```

注意

如果经常使用唯一的结果值，其依附的属性也是基于唯一值的，你可以考虑 natural-id（自然 ID）的映射，并用 natural-id（自然 ID 加载）API。参考 ***Hibernate Domain Mapping Guide（Hibernate 域映射手册）*** 了解更多 natural-id（自然 ID）的信息。

Hibernate 提供 2 个额外的专用方法执行查询、处理结果。Query#scroll 与 JDBC 滚动结果集协同工作。scroll 方法被重载过。第一个方法接受单一参数，

类型为 org.hibernate.ScrollMode，参数表示滚动使用的类型。参考 javadoc 的滚动模式了解细节。第二个方法没有参数，表示滚动将使用 Dialect#defaultScrollMode（方言的默认类型）。Query#scroll 返回 org.hibernate.ScrollableResults。它封装底层 JDBC（可滚动）结果集，通过它可以对结果进入访问。由于这种形式控制 JDBC 结果集的打开。应用程序使用完滚动结果集时应显式调用结果集的 close 方法（从 java.io.Closeable 继承得到，因此滚动结果集要放到 Try 块中）。如果应用程序没有调用关闭方法。Hibernate 将在事务结束时自动关闭结果集。

注意

如果你计划使用 Query#scroll 处理集合抓取。要注意了，你的查询要显式排序，这样 JDBC 结果包含的相关行才有顺序。

最后是 Query#iterate，当应用程序一定从二级缓冲中加载实体时要考虑使用这种方式。背后的思想是，迭代必须匹配标识符，这些标识符是从 SQL 查询中得到的。因为这些标识符被二级缓冲查找、解析。如果二级缓冲是没有找到，额外的查询将向数据库发出。如果实体已经在二级缓冲中存在了，那么这个操作是很好的加载实体方法。但是如果很多实体没有存在于二级缓冲中，这个操作很坏的方法。Query#iterate 返回的迭代器实际的类型是 org.hibernate.engine.HibernateIterator。它专门暴露了 close 方法（从 java.io.Closeable 继承得到）。当我们使用完这个迭代器时，我们应该关闭它，或者抛给 HibernateIterator，或抛给 Closeable ，或者调用 org.hibernate.Hibernate#close 方法。

### 13.10.2. JPA 查询 API

在 JPA 中，查询是从 EntityManager 中得到的 javax.persistence.Query 或 javax.persistence.TypedQuery。如果是命名查询使用 EntityManager#createNamedQuery 方法，其它查询使用 EntityManager#createQuery 方法。

**示例 13.47. 获取查询引用--JPA**

```
Query query = em.createNamedQuery( "my-predefined-named-query" );
```

```
        TypedQuery<MyEntity> query2 = em.createNamedQuery(
                "my-predefined-named-query",
                MyEntity.class
        );
Query query = em.createQuery(
                "select e from MyEntity e where name like :filter"
        );
        TypedQuery<MyEntity> query2 = em.createQuery(
                "select e from MyEntity e where name like :filter"
                MyEntity.class
        );
```

注意

这一切听起来很熟悉。不仅 JPQL 语法受 HQL 启发，而且 JPA API 也受 Hibernate 启发。以下 2 个查询很熟悉。

这查询接口能够控制查询的执行。例如，我们能够指定执行超时或控制缓冲。

**示例 13.48. 基本查询用法--JPA**

```
Query query = ...;
        // timeout - in milliseconds
        query.setHint( "javax.persistence.query.timeout", 2000 )
        // Do not peform (AUTO) implicit flushing
        query.setFlushMode( COMMIT );
```

了解全部细节，参考 javadoc 的查询内容。很多设置由定义 hint（提示）来控制查询执行。JPA 定义一些标准 hint（提示）（比如示例中的超时），但是更多的是专用的。由于这些专用的 hint(提示)，在一定程度上限制你应用程序的可移植性。

**JPA 标准查询 hint（提示）**

javax.persistence.query.timeout--定义查询超时时间，单位是毫秒 。

javax.persistence.fetchgraph--定义"fetchgraph" EntityGraph（实体图）。属性显式指定 FetchType.EAGER（通过 join 抓取或者子查询抓取）。更多细节参考实体图讨论第九章 *抓取*。

javax.persistence.loadgraph--定义"loadgraph"EntityGraph（实体图）。属性显式指定 FetchType.LAZY 或 FetchType.EAGER（通过 join 抓取或者子查询抓取）。属性没有指定 FetchType.LAZY 或 FetchType.EAGER 时依赖属性的 metadata 中的定义。更多细节参考实体图讨论第九章 *抓取*。

### Hibernate 指定 JPA 查询 hint

org.hibernate.cacheMode--定义缓冲模式，参考 org.hibernate.Query#setCacheMode。

org.hibernate.cacheable--定义是否使用缓冲，true/false。参考 org.hibernate.Query#setCacheable。

org.hibernate.cacheRegion--为可缓冲的查询定义缓冲区，参考 org.hibernate.Query#setCacheRegion。

org.hibernate.comment--为生成的 SQL 定义批注。参考 org.hibernate.Query#setComment。

org.hibernate.fetchSize--定义 JDBC 抓取大小。参考 org.hibernate.Query#setFetchSize。

org.hibernate.flushMode--定义抓取模式。参考 org.hibernate.Query#setFlushMode。如果可能，最好使用 javax.persistence.Query#setFlushMode 替代。

org.hibernate.readOnly--定义实体与集合以只读方式加载。参考 org.hibernate.Query#setReadOnly。

正如 Hibernate API 看到的，执行查询前做的最后一件事是为查询中定义的参数绑定值。JPA 定义一些简单参数绑定方法。它支持设置参数值（通过名称/位置），并且定义了一个特殊的日历/时间类型 TemporalType。

**示例 13.49. 参数绑定--JPA**

```
Query query = em.createQuery(
                "select e from MyEntity e where e.name like :filter"
```

```
        );
        query.setParameter( "filter", "D%" );

        Query q2 = em.createQuery(
                "select e from MyEntity e where
e.activeDate > :activeDate"
        );
        q2.setParameter( "activeDate", new Date(),
TemporalType.DATE );
```

另外，JPA 也允许访问有关参数的信息。

至于执行，正如上面的 Hibernate API 一样，JPA 支持两个主要的方法。分别是调用 Query#getResultList 方法和 Query#getSingleResult 方法。他们行为与上面介绍的 org.hibernate.Query#list 方法和 org.hibernate.Query#uniqueResult 方法一样。

## 第 14 章 Criteria

**目录**

## 14.5. 使用参数

Criteria 查询（译者注：网上有说叫条件查询，动态查询，标准查询） 提供类型安全的 hql，JPQL 与 native-sql 查询。

注意

Hibernate 提供老的，过时的 org.hibernate.CriteriaAPI，这些 API 已经过时了，没有发展。最终 Hibernate 专用的 criteria 的功能将移植到 JPA 的 javax.persistence.criteria.CriteriaQuery。更多细节在 org.hibernate.CriteriaAPI 中，参考附录 B,*Legacy（过时的） Hibernate Criteria 查询*

本章将聚焦在使用 JPA 的 API 进行声明类型安全的 criteria 查询。

Criteria 查询是一种编程式，类型安全的查询方式。Criteria 使用接口与类表示查询的各种结构，比如：查询本身，select 从句，排序操作等等，这些都是类型安全的。正如我们看到的，它们也可以类型安全的引用属性。JPA API 设计的很好，老 org.hibernate.Criteria 的用户可以很方便的识别一般的方法。Criteria 查询本质上是一个对象图，图的每一部分代表增量（我们按图进行导航）。执行查询的第一步是建立对象图。

javax.persistence.criteria.CriteriaBuilder 接口是第一件事，代表我们开始使用 Criteria 查询，它的角色是一个 criteria 所需内容的工厂。你通过 javax.persistence.EntityManagerFactory 或者 javax.persistence.EntityManager 调用 getCriteriaBuilder 方法得到 javax.persistence.criteria.CriteriaBuilder 的实例。

下一步是得到 javax.persistence.criteria.CriteriaQuery。使用以下 3 方法之一完成这一目的，这三个方法都是 javax.persistence.criteria.CriteriaBuilder 的方法：

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

每一个服务有不同的目的，选择哪个服务是根据查询结果集的预期类型。

注意

JPA 说明书*第六章 Criteria API* 已经包含了大量的有关 criteria 查询各个方面的参考资料。所以与其在此复制所有内容不如让我们关注一下更广泛的 API 以及未来的使用方法。

## 14.1. 类型化 criteria 查询

criteria 查询的类型（也叫<T>）是指查询结果期望的类型。这可能是一个实体，或一个 Integer,或任何其它的对象。

### 14.1.1. 选择一个实体

这可能是最常用的查询形式。应用程序想要 select 一个实体实例。

**示例 14.1. 选择一个 root(根)实体**

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

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

示例使用 createQuery 传入 Person 类引用，查询结果将是 Person 对象。

注意

这个例子中调用 CriteriaQuery.select 方法不是是必要的，因为我们只有一个单一的查询 root(根)，所以 *personRoot* 可以隐式选择。示例中放上是为了给大家一个完整的示例。

上例中的 *Person_.eyeColor* 引用一个 JPA metamodel 引用的静态模型。我们将在本章中只使用这个模型。更多的细节参考 Hibernate JPA Metamodel Generator 中的 JPA static metamodel。

### 14.1.2. 选择一个表达式

选择一个表达式最简单的形式就是选择实体的一个特定属性的模式。但是这个表达式也可能表示一个聚合、一个算术运算符，等等。

**示例 14.2. 选择一个属性**

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

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

在此示例中，查询类型是 java.lang.Integer，因为这是一个预先知道的类型（Person#age 属性的类型是 java.lang.Integer）。因为查询可能包含 Person 实体的多个引用（属性），所以属性引用一定是加上限制。这通过调用 Root#get 方法来实现。

### 14.1.3. 选择多个值

实际上使用 criteria 查询时有多种途径选择多个值。我们探索两种方法，但是另一种推荐的方法是使用 tuple（元组）在 14.2, "Tuple criteria 查询"描述，或者考虑 Wrapper(封装)查询，参考 14.1.4, "选择 wrapper(封装)"了解细节。

**示例 14.3. 选择一个数组**

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

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

从技术上讲，这个应该归类为类型查询，但是你从处理结果中可以看到这有点误导。总之，期望的结果类型是一个数组。

示例中使用了 javax.persistence.criteria.CriteriaBuilder 的 array 方法，此方法显式组合多个独立的选项进入 javax.persistence.criteria.CompoundSelection。

**示例 14.4. 选择一个数组（2）**

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

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

```
        }
```

正如你所看到的，示例 14.3,"选择一个数组"我们有一个返回对象数组的类型化 criteria 查询。这两个查询功能上是等价的。第二个示例使用 multiselect 方法，这与第一个有一点不同，就是 criteria 查询刚开始建立时提供类型的方式不同。第二种情况下也可以说是选择并返回了一个 *Object[]*（译者注：第一个是类型查询，只不过类型是一个数组；第二个是一个查询，返回的是一个数组）。

#### 14.1.4. 选择 wrapper（封装）

另一种替代 14.1.3,"选择多个值"的方法是选择一个"wrap（封装）"了多个值的封装对象。回到示例中，不再返回*[Person#id, Person#age]*的数组了，取而代之的返回一个声明好的类，这个类中包含这些值。

**示例 14.5. 选择一个封装**

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

        ...

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

```
        );
        criteria.where( builder.equal( personRoot.get( Person_.eyeCol
or ), "brown" ) );

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

我们看到首先声明一个简单的封装对象，我们将使用这个封装对象封装我们期望的结果值。特别注意封装对象的构造函数与参数类型。因为我们要返回 PersonWrapper 对象，所以我们使用 PersonWrapper 作为 criteria 查询的类型。这个示例说明了 javax.persistence.criteria.CriteriaBuilder 的 construct 方法如何使用，它将建立一个封装表达式。这个封装表达式的每一行都是我们要表达的意思，*PersonWrapper* 通过构造函数实例化，构造函数的参数要与封装过程传递内容相符合。这个封装表达式会传递给 select（译者注：上例中有一个双参构造，builder.construct 调用这个构造，并传入正确的参数）。

## 14.2. Tuple criteria 查询

比 14.1.3,"选择多个值"更好的方法是使用封装（正如我们看到的 14.1.4,"选择一个封装"），或者使用 javax.persistence.Tuple 约定。

**示例 14.6. 选择一个 tuple**

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

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

```
            assert tuple.get( 1 ) == tuple.get( agePath );
            ...
        }
```

这个示例通过 javax.persistence.Tuple 接口访问查询结果。本示例中显式调用 javax.persistence.criteria.CriteriaBuilder 的 createTupleQuery 方法，另一种替代方案是使用 createQuery，并传入一下 Tuple.class 参数。
我们再次看到了 multiselect 方法，正如示例 14.4,"选择一个数组(2)"中一样。不同之处是 javax.persistence.criteria.CriteriaQuery 的类型被定义为 javax.persistence.Tuple，然后数组被 tuple 元素取代。
javax.persistence.Tuple 约定提供 3 种形式访问底层的元素：
typed（类型）

示例 14.6,"选择一个 tuple"中 tuple.get(idPath) 与 tuple.get(agePath)的调用，说明了这种访问形式。这种访问是基于 javax.persistence.TupleElement 表达式，这个表达式是在建立 Criteria 时在底层构建的，我们可以对它的 tuple 值进行访问。
positional（位置）

可以基于位置对于底层 tuple 值的访问。简单使用 *Object get(int position)*的形式，这非常类似于示例 14.3,"选择一个数组"和示例 14.4,"选择一个数组(2)"的访问方式。另一种位置访问形式是*<X> X get(int position, Class<X> type)*，但是使用些种方式，要求提供类型，所以 tuple 中值的类型一定是可类型化的。
aliased（别名）

可以使用基于别名（可选）对于底层 tuple 值的访问。示例中没有应用别名。别名是通过 javax.persistence.criteria.Selection 的 alias 方法定义的，与 positional（位置）访问一样也同样有两种访问方法：已经定义过类型的用 *Object get(String 别名)*；没有定义过类型的用*<X> X get(String 别名, Class<X> type)*。

## 14.3. FROM 子句

注意

FROM 子句的所有个体（roots,joins,paths）都实现了

javax.persistence.criteria.From 接口。

### 14.3.1. Roots(根)

根的定义是所有查询可用的 joins，paths，attributes（join，路径，属性）的基础。根永远是一个实体类型。根通过 javax.persistence.criteria.CriteriaQuery 接口重载的 from 方法进行定义与添加：

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

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

**示例 14.7. 添加根 Adding a root**

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

Criteria 查询可能定义多个根，效果就象是新加入的根与其它的根之间建立一个笛卡尔乘积。这里有一个示例，配对所有的单身男人与所有的单身女人。

**示例 14.8. 添加多个根**

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

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

        builder.equal( women.get( Person_.relationshipStatus ),
RelationshipStatus.SINGLE )
        );
        query.where( builder.and( menRestriction,
womenRestriction ) );
```

### 14.3.2. Joins

Join 允许从 javax.persistence.criteria.From 到其它联合或者嵌入属性的导航。Join 是由 javax.persistence.criteria.From 接口中大量重载的 join 方法建立的。

**示例 14.9. 使用插件属性与多对一**

```
CriteriaQuery<Person> personCriteria =
builder.createQuery( Person.class );
        Root<Person> personRoot = person.from( Person.class );
        // Person.address 是一个嵌入的属性
        Join<Person,Address> personAddress =
personRoot.join( Person_.address );
        // Address.country 是多对一
        Join<Address,Country> addressCountry =
personAddress.join( Address_.country );
```

**示例 14.10. 集合示例**

```
CriteriaQuery<Person> personCriteria =
builder.createQuery( Person.class );
        Root<Person> personRoot = person.from( Person.class );
```

```
        Join<Person,Order> orders = personRoot.join( Person_.orders );
        Join<Order,LineItem> orderLines =
orders.join( Order_.lineItems );
```

### 14.3.3. 抓取

如同 HQL 与 JPQL 一样，criteria 查询可以指定关联的数据随所有者一起抓取。抓取是 javax.persistence.criteria.From 接口的 fetch 方法，此方法有多种重载形式。

**示例 14.11. 插件属性与多对一**

```
CriteriaQuery<Person> personCriteria =
builder.createQuery( Person.class );
        Root<Person> personRoot = person.from( Person.class );
        // Person.address 是一个嵌入的属性
        Fetch<Person,Address> personAddress =
personRoot.fetch( Person_.address );
        //Address.country 是多对一
        Fetch<Address,Country> addressCountry =
personAddress.fetch( Address_.country );
```

注意

从技术上讲，嵌入属性总是与其所有者一起抓取的，然而为了定义明确的抓取 *Address#country*，我们需要定义 javax.persistence.criteria.Fetch 的路径。

**示例 14.12. 集合示例**

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

## 14.4. 路径表达式

注意

Roots,joins，fetches（根，join 与抓取）本身的路径也是如此(译者注：路径可以用于 root ,join,fetch 操作)。

## 14.5. 使用参数

**示例 14.13. 使用参数**

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

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

使用 javax.persistence.criteria.CriteriaBuilder 的 parameter 方法得到参数的引用，然后使用参数引用绑定参数的值到 javax.persistence.Query 中。

# 第 15 章 Native SQL 查询（原生 SQL 查询，本地 SQL 查询）

**目录**

你也可以使用你数据库的 native SQL（原生 SQL，本地 SQL）方言进行查询。如果你想利用数据库特性比如 Oracle 特有的查询的 hint 或者 CONNECT BY 操作时，native SQL 非常有用。这也提供了一个完整迁移路径，从基本的 SQL/JDBC 应用迁移到 Hibernate/JPA 应用。Hibernate 也允许你手写 SQL（包括存储过程），这些手定的 SQL 可以用于创建、更新、删除与加载操作。

## 15.1. 使用 SQLQuery

通过 SQLQuery 接口控制 native SQL 的执行，SQLQuery 是通过调用 Session.createSQLQuery()得到的。以下章节讨论如何使用这些 API。

### 15.1.1. Scalar（标量）查询

最基本的 SQL 查询是得到一个 Scalar（标量）（值）列表

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

这里返回一个 CATS 表中每列的标量值的对象值列表（Object[]）。Hibernate 将使用 ResultSetMetadata 推断出返回标量值的实际顺序与类型。
为了避免 ResultSetMetadata 的开销，或只是简单的为了显式指明返回什么，可以使用 addScalar()方法：

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

这查询指定了：
SQL 查询字符串
返回的列与类型
这将返回对象数组。但是现在不是使用 ResultSetMetadata，而是从底层的结果集中显式得到 ID，NAME，BIRTHDATE 列，类型分别为 Long，String ，短日期。这就意味返回的只有这三列。即使查询使用*或者超过 3 列。
省略所有或部分的标量信息是有可能的。

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

这个本质上与上面的例子是一样的查询，但是现在使用 ResultSetMetaData 确认 NAME 与 BIRTHDATE 的类型， 但是 ID 的类型已经显示指定了。
ResultSetMetaData 返回的 java.sql.Types 会被映射成 Hibernate 类型，这是在方言控制下完成的。如果指定的类型没有映射或者没有找到期望的结果，方言可以调用 registerHibernateType 自定义类型。

### 15.1.2. 实体查询

上面的查询都返回的标题标量值，从结果集中返回的基本上是"原始的"值。下面演示从 native SQL 如何得到实体对象，这要通过 addEntity()方法。

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

这个查询指定了：
查询字符串
查询返回的实体
假设 Cat 是一个映射类，上面查询的列（ID，NAME ，BIRTHDATE）将返回一个列表，每一行都是一个 Cat 实体。
如果实体映射中有 many-to-one，当执行 native（原生、本地）查询时另一个实体也需要返回，否则将发生数据库特定的“column not found”错误。当使用 * 标记时其它列将自动返回。但是我们更希望显式使用下面的示例，返回 many-to-one 关联的 Dog 实体：

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

这将允许执行 cat.getDog()功能返回 Dog 属性。

### 15.1.3. 处理 association（关联）与集合

为了避免可能出现往返的初始化过程，可能想 eagerly join(立即连接)到 Dog 实体。这可以通过 addJoin()方法，这个方法允许你连接到关联（译者注：多对一）或集合（译者注：一对多）。

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

在示例中，返回了 Cat 与它们的 dog 属性，这没有其它往返的数据库操作，只在一次初始化中完成了。注意在这里我们还添加了一个别名("cat")，这个别名的作用是连接到目标属性时生成连接路径（译者注："cat.dog"）。也可以将这种 eager(立即)连接应用到集合上，例如，如果 Cat 替换成一个 one-to-many（一对多）的 Dog。

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

在这个阶段，你可能已经达到了 native 查询的极限了，native SQL 不可能像 Hibernate 的 HQL 一样有那么多增强的功能。下面两种情况下可以有问题出现：同时返回同一类型的多个实体（译者注：如一个猫中有两种狗的关联）；默认的别名、列名无法使用（译者注：可能与其它别名有冲突时）。

### 15.1.4. 返回多个实体

到目前为止，一直假定结果集的列名与映射文档的列名是相同的。当 SQL 查询连接多个表是，这可能会有问题，因为相同的列名会出现在多个表中。
下面的查询中需注入列别名（这很可能失败）：

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

查询期望在一行中返回两个 Cat 实体：猫与它母亲。然而查询将失败，因为名称冲突；实体映射相同列的名称。从数据库返回的列别名如"c.ID", "c.NAME"等等。它们在映射文件中是（"ID"与"NAME"），但是在本示例中不是唯一的。
下面的形式在列名重复的情况下是不容易冲突的：

```
        //译者注：原文可能有误下面的代码可能是{cat.*}, {mother.*}
        sess.createSQLQuery("SELECT {cat.*}, {m.*}  FROM CATS c, CATS
m WHERE c.MOTHER_ID = m.ID")
         .addEntity("cat", Cat.class)
         .addEntity("mother", Cat.class)
```

这个查询指定了:
查询字符串，用占位符为 Hibernate 注入列别名
查询返回的实体
上面的{cat.*} 与 {mother.*} 标记是“所有属性”的简写。做为一种选择，你可以显式列出所有列，但是即使这样 Hibernate 也还会为每个属性注入列别名。列别名的占位符正好是表别名+属性名。（译者注：不指定别名 hibernate 自动加一个，如果指定别名 hibernate 就加上你指定的别名）下面的示例，你从另一个表(cat_log)检索猫与它们母亲，但是映射 metadata 与上面的示例是一个。你甚至可以在 where 子句使用属性别名。

```
        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()
```

#### 15.1.4.1. 别名与引用属性

在大多数情况下，上面示例中的别名是必需的。对于查询涉及到更复杂的映射关系时，比如联合属性，继承的鉴别器，集合等等，你可以使用具体的别名，这样就可以允许 hibernate 注入合适的别名了。
下面的表格展示使用别名注入的不同方式。请注意结果中的别名只是一个简单示例。当使用别名时，每个别名必须是唯一的且不同的。

**表 15.1. 别名注入命名**

### 15.1.5. 返回非托管实体

应用一个 ResultTransformer（结果转换器）在一个 native SQL 查询上是可能的，这果就有可能返回一个非托管的实体。

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

这个查询指定了
SQL 查询字符串

一个结果转换器
上面的查询将返回一个 CatDTO 列表，这个列表的实体将自动实例化并且注入 NAME，IRTHNAME 值到相对应的属性或字段中。

### 15.1.6. 处理继承

Native SQL 查询，当查询被映射为一个继承的实体时，这个查询必须包括所有父类的属性与所有子类的属性。

### 15.1.7. 参数

Native SQL 查询支持定位参数和命名参数：

```
        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();
```

## 15.2. 命名 SQL 查询

命名 SQL 查询可以定义在映射文档中，调用过程与命名 HQL 查询是完全一样的。在这种情况下，你不必调用 addEntity()方法。

**示例 15.1. 命名 SQL 查询使用的映射<sql-query>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>
```

**示例 15.2. 执行命名查询**

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

<return-join>元素用于连接关联，<load-collection>元素用于定义带有初始化集合的查询，

**示例 15.3. 带有关联的命名 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>
```

命名 SQL 查询可以返回 scalar(标量)值。你可以使用<return-scalar>元素声明列的别名与 Hibernate 类型：

**示例 15.4. 返回标量的命名查询　Named query returning a scalar**

```
<sql-query name="mySqlQuery">
                <return-scalar column="name" type="string"/>
                <return-scalar column="age" type="long"/>
                SELECT p.NAME AS name,
                           p.AGE AS age,
                FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
        </sql-query>
```

你可以将结果集映射在外部进行定义，这可以通过<resultset>元素完成，这样好处有两方面：1.多个命名查询可以重用相同的结果集；2.通过setResultSetMapping()方法动态加载。

**示例 15.5. 使用外部<resultset>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>
```

你还可使用将结果集映射信息写入 hbm 文档，然后在 java 代码中直接使用。

**示例 15.6. 以编程的方式指定结果映射信息**

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

到目前为至，我们只是看到在 Hibernate 映射文档使用外部的 SQL 查询。相同的概念使用注释也是有效的，并且可以通过命名 native 查询调用。你可以使用 @NamedNativeQuery(@NamedNativeQueries)连接上 @SqlResultSetMapping(@SqlResultSetMappings)。就像@NamedQuery 一样使用。@NamedQuery,@NamedNativeQuery，@SqlResultSetMapping 可以定义在代码的类中，但是它们的作用域是全局的。让我们看下面的例子。

示例 15.7,"使用@NamedNativeQuery 连用@SqlResultSetMapping 进行命名 SQL 查询"展示了@NamedNativeQuery 如何定义 resultSetMapping 参数。这个参数的值是由@SqlResultSetMapping 定义的映射名。native 查询检索出结果集映射定义的实体。实体的每个字段都绑定着 SQL 别名（或列名）。实体的字段中，子类、关联的实体的外键必须存在于 SQL 查询中。字段定义是可选的，如果不定义，它们映射到与类属性声明中有相同名称的列名上。在示例 2 中返回了两个实体 Night 与 Area，它们的每个属性都关联到了实际查询返回的列名上。

示例 15.8,"隐式结果集映射"中，结果集映射是隐式的。我们只是描述了结果集映射与实体类的关系（译者注：@EntityResult）。属性到列名的映射由实体类中定义的映射值完成（译者注：@Column(name="model_txt")），在这种情况下，model 属性绑定到了 model_txt 列上。

最后，如果关联实体有联合主键，@FieldResult 元素必须使用在每一个外键列上。@FieldResult 命名是由以下内容组合：相关联的属性名+“.”+字段或主键属性名（译者注：参考示例中的 name="captain.firstname"）。这些可以看示例 15.9,"在指定关联的@FieldResult 中使用点标记符"。

**示例 15.7. 使用@NamedNativeQuery 连用@SqlResultSetMapping 进行命名 SQL 查询**

```
@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.are
a_id = area.id",
                                        resultSetMapping="joinMap
ping")
        @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.annotati
ons.query.Area.class, fields = {
                        @FieldResult(name="id", column="aid"),
                        @FieldResult(name="name", column="name")
                })
                }
        )
```

**示例 15.8. 隐式结果集映射**

```
@Entity
        @SqlResultSetMapping(name="implicit",
                                        entities=@EntityRes
ult(entityClass=SpaceShip.class))
        @NamedNativeQuery(name="implicitSample",
                                        query="select * from S
paceShip",
                                        resultSetMapping="implici
t")
        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;
                }
        }
```

**示例 15.9. 在指定关联的@FieldResult 中使用点标记符**

```
@Entity
        @SqlResultSetMapping(name="compositekey",
                        entities=@EntityResult(entityClass=SpaceShip.
class,
                                fields = {
                                        @FieldResult(name="na
me", column = "name"),
                                        @FieldResult(name="mo
del", column = "model"),
                                        @FieldResult(name="sp
eed", column = "speed"),
                                        @FieldResult(name="ca
ptain.firstname", column = "firstn"),
```

```
                                                @FieldResult(name="captain.lastname", column = "lastn"),
                                                @FieldResult(name="dimensions.length", column = "length"),
                                                @FieldResult(name="dimensions.width", column = "width")
                                                }),
                        columns = { @ColumnResult(name = "surface"),
                                                @ColumnResult(name = "volume") } )

        @NamedNativeQuery(name="compositekey",
                query="select name, model, speed, lname as lastn, fname as firstn, length, width, length * width as surface from SpaceShip",
                resultSetMapping="compositekey")
        } )
        public class SpaceShip {
                private String name;
                private String model;
                private double speed;
                private Captain captain;
                private Dimensions dimensions;

                @Id
                public String getName() {
                        return name;
                }

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

                @ManyToOne(fetch= FetchType.LAZY)
                @JoinColumns( {
                                @JoinColumn(name="fname", referencedColumnName = "firstname"),
                                @JoinColumn(name="lname", referencedColumnName = "lastname")
                                } )
                public Captain getCaptain() {
                        return captain;
                }
```

```
		public void setCaptain(Captain captain) {
			this.captain = captain;
		}

		public String getModel() {
			return model;
		}

		public void setModel(String model) {
			this.model = model;
		}

		public double getSpeed() {
			return speed;
		}

		public void setSpeed(double speed) {
			this.speed = speed;
		}

		public Dimensions getDimensions() {
			return dimensions;
		}

		public void setDimensions(Dimensions dimensions)
{
			this.dimensions = dimensions;
		}
	}

	@Entity
	@IdClass(Identity.class)
	public class Captain implements Serializable {
		private String firstname;
		private String lastname;

		@Id
		public String getFirstname() {
			return firstname;
		}

		public void setFirstname(String firstname) {
			this.firstname = firstname;
```

```
                }

                @Id
                public String getLastname() {
                        return lastname;
                }

                public void setLastname(String lastname) {
                        this.lastname = lastname;
                }
        }
```

建议

如果你使用默认映射只检索单一实体，你可以指定 resultClass 属性代替 resultSetMapping：

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

在一些 native 查询中，你不得不返回标题值，例如建立一个 reportquery(报表查询)。你可以在@SqlResultsetMapping 中使用@ColumnResult 映射。实际上你甚至可以在同一个 native 查询的返回中混合实体与标量（虽然这不是很常用）。

**示例 15.10. 通过@ColumnResult 得到 scalar(标量)值**

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

另一个 native 查询的特殊提示已经介绍过了：org.hibernate.callable 的值为 true 或 false 决定是否使用存储过程。

### 15.2.1. 使用 return-property 显式指定列与别名

你可以显式告诉 Hibernate 列的别名是什么，这是通过使用<return-property>标记完成，这个标记代替{}的语法让 Hibernate 注入自己的别名。例如：

```
<sql-query name="mySqlQuery">
                <return alias="person" class="eg.Person">
                        <return-property name="name" column="myName"/>
                        <return-property name="age" column="myAge"/>
                        <return-property name="sex" column="mySex"/>
                </return>
                SELECT person.NAME AS myName,
                           person.AGE AS myAge,
                           person.SEX AS mySex,
                FROM PERSON person WHERE person.NAME LIKE :name
        </sql-query>
```

<return-property>也可以应用于多列属性。这解决了{}语法的限制。{}不能精确控制多列属性。

```
<sql-query name="organizationCurrentEmployments">
                <return alias="emp" class="Employment">
                        <return-property name="salary">
                                <return-column name="VALUE"/>
                                <return-column name="CURRENCY"/>
                        </return-property>
                        <return-property name="endDate"
column="myEndDate"/>
                </return>
                        SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS
{emp.employer},
                        STARTDATE AS {emp.startDate}, ENDDATE AS
{emp.endDate},
                        REGIONCODE as {emp.regionCode}, EID AS
{emp.id}, VALUE, CURRENCY
                        FROM EMPLOYMENT
                        WHERE EMPLOYER = :id AND ENDDATE IS NULL
                        ORDER BY STARTDATE ASC
        </sql-query>
```

在示例中，<return-property>与{}语法组合使用完成注入。这就允许用户选择怎样引用列与属性。

如果你的映射中有 discriminator（鉴别器），你必须使用<return-discriminator>指定鉴别器列。

### 15.2.2. 使用存储过程查询

Hibernate 支持通过存储过程或函数进行查询。下面文档中两者是等价的。存储过程或函数必须以结果集作为第一个外传的参数（out-parameter），这样才能与 Hibernate 协同工作。示例中存储过程与和函数可以运行在 Oracle 9 或更高的版本中：

```
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;
```

在 Hibernate 中使用这样的查询，需要映射为命名查询。

```
<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>
```

存储过程目前只能返回标量与实体。不支持<return-join>与<load-collection>。

#### 15.2.2.1. 使用存储过程的规范与限制

除非你遵守下面的规范，否则不能在 Hibernate 中使用存储过程。如果你不遵守规范又想用 Hibernate，你可以用 session.connection()直接执行。不同的数据库这些规范是不同的，因为不同数据库供应商有不同的存储过程语义和语法。

存储过程查询不能使用分页功能的函数 setFirstResult()/setMaxResults()。

推荐的调用形式是标准 SQL92 的：

{ ? = call functionName(<parameters>) }或

{ ? = call procedureName(<parameters>}。native(原生)的调用语法不支持。

对于 Oracle 应用下列规范

函数必须返回结果集，存储过程的第一个参数必须是 OUT 的结果集。结果集可以使用 Oracle 9 或 10 的 SYS_REFCURSOR 类型。在 Oracle 中你必须定义 REF CURSOR 类型。参考 Oracle 的相关文献了解更多信息。

对于 Sybase 或 MS SQL server 应用下列规范：

存储过程只能返回一个结果集。注意，由于服务器支持返回多个结果集与更新计数器，Hibernate 只迭代第一个结果集，其它内容将抛弃。

过程中你可以使用 SET NOCOUNT ON 以提高效率，但这不是必须的条件。

## 15.3. 自定义新建、更新、删除 的 SQL 语句

Hibernate 允许自定义 SQL 执行新建、更新、删除操作。SQL 可以在 statement level（语句级）重写或是 column level（列级）重写。本节描述是语句级的重写。关于列级的重写参考这里。 示例 15.11,“通过注释自定义 CRUD”展示如何使用注释使用自定义的 SQL 操作。

**示例 15.11. 通过注释自定义 CRUD**

```
@Entity
        @Table(name="CHAOS")
```

```
        @SQLInsert( sql="INSERT INTO CHAOS(size, name, nickname,
  id) VALUES(?,upper(?),?,?)")
        @SQLUpdate( sql="UPDATE CHAOS SET size = ?, name = u
pper(?), nickname = ? WHERE id = ?")
        @SQLDelete( sql="DELETE CHAOS WHERE id = ?")
        @SQLDeleteAll( sql="DELETE CHAOS")
        @Loader(namedQuery = "chaos")
        @NamedNativeQuery(name="chaos", query="select id, size, n
ame, lower( nickname ) as nickname from CHAOS where id= ?",
  resultClass = Chaos.class)
        public class Chaos {
                @Id
                private Long id;
                private Long size;
                private String name;
                private String nickname;
```

@SQLInsert,@SQLUpdate,@SQLDelete,@SQLDeleteAll 分别重写 INSERT,UPDATE,DELETE,与 DELETE all 语句。同样的操作可以使用 Hibernate 映射文件与<sql-insert>，<sql-update>，<sql-delete>节点实现。这个参考 示例 15.12,"xml 文件中自定义 CRUD "

**示例 15.12. xml 文件中自定义 CRUD**

```
<class name="Person">
                <id name="id">
                        <generator class="increment"/>
                </id>
                <property name="name" not-null="true"/>
                <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES
( UPPER(?), ? )</sql-insert>
                <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE
ID=?</sql-update>
                <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-
delete>
        </class>
```

如果你们期望调用存储过程，确保设置 callable 属性为 true。注释与 xml 是一样的。

如果想检查执行是否正确，Hibernate 允许你使用下面三种策略之一：

none:不执行检查；如果出现问题存储过程执行失败。

count:使用行数检查 update 是否成功

param:像 COUNT 一样，但是使用 output 参数，不是标准的机制。

要定义检查风格，使用 check 参数，注释与 xml 文件一样。

你重写与集合相关的语句，注释与 xml 节点是相同设置--参考示例 15.13,"使用注释重写集合有关的 SQL 语句"

**示例 15.13. 使用注释重写集合有关的 SQL 语句**

```
@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<Cas
imirParticle>();
```

提示

参数的顺序很重要，参数的顺序是由 Hibernate 处理属性的顺序确定的。通过将调试日志的级别定义为 org.hibernate.persister.entity，你就可以查看期望的顺序。在这种级别下，Hibernate 将打印出建立、更新、删除实体时执行的静态 SQL 语句。（查看期望的顺序，记住不包括你通过注释与 XML 文件自定义的 SQL，因为它们已经重写了 Hiebernate 生成的静态 SQL）。

重写 SQL 语句实现还可以实现 secondary table（从表）功能，使用 @org.hibernate.annotations.Table 标记加上 sqlInsert,sqlUpdate,sqlDelete 属性的全部或一部分：

**示例 15.14. 重写 SQL 语句实现 secondary tables（从表）**

```
@Entity
        @SecondaryTables({
                @SecondaryTable(name  =  "`Cat  nbr1`"),
```

```
                @SecondaryTable(name = "Cat2")})
        @org.hibernate.annotations.Tables( {
                @Table(appliesTo = "Cat", comment = "My cat table" ),
                @Table(appliesTo = "Cat2", foreignKey = @ForeignKey(name="FK_CAT2_CAT"), fetch = FetchMode.SELECT,
                        sqlInsert=@SQLInsert(sql="insert into Cat2(storyPart2, id) values(upper(?), ?)") )
        } )
        public class Cat implements Serializable {
```

上面的示例还展示了 comment（注释）功能，你可以使用 comment（注释）传给表（主表或从表）：这些 comment（注释）将在建立 DDL 时使用。

建议

这些 SQL 是直接在数据库中执行的，因此你可以使用任意的你喜欢的方言。但是，如果使用了特定的 SQL，会降低可移植性。

最后但并非不重要，存储过程大多数情况下返回添加、更新、删除操作影响的行数。Hibernate 对于 CUD 操作，总是将第一个输出参数定义为数值（译者注：没有 R 操作，如果是 R 操作第一个 out 参数是结果集）：

**示例 15.15. 存储过程与返回值**

```
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;
```

## 15.4. 自定义 SQL 加载

你可以声明你自己的 SQL(或者 HQL)查询用于实体加载。添加，更新，删除这些操作可以定义在具体列级别或 statement（语句）级别。这有一个在语句级别重写的示例：

```
<sql-query name="person">
                <return alias="pers" class="Person" lock-
mode="upgrade"/>
                SELECT NAME AS {pers.name}, ID AS {pers.id}
                FROM PERSON
                WHERE ID=?
                FOR UPDATE
        </sql-query>
```

这只是声明了一个命名查询，如前讨论的一样，你可以在类映射中引用这个命名查询：

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

这些甚至可以与存储过程协同工作。

你也可以为集合加载定义查询：

```
<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>
```

你也可以定义一个实体加载器通过 join 抓取加载一个集合：

```
<sql-query name="person">
```

```
                <return alias="pers" class="Person"/>
                <return-join alias="emp"
property="pers.employments"/>
                SELECT NAME AS {pers.*}, {emp.*}
                FROM PERSON pers
                LEFT OUTER JOIN EMPLOYMENT emp
                        ON pers.ID = emp.PERSON_ID
                WHERE ID=?
        </sql-query>
```

xml 中<loader>元素与注释的@Loader 是等价的，参考示例 15.11,“通过注释自定义 CRUD”

## Chapter 16. Multi-tenancy（多租户）

**目录**



### 16.1. 什么是 Multi-tenancy（多租户）

术语 multi-tenancy（多租户）通常是指软件开发的一种体系结构，在这种体系结构中：一个应用程序的运行实例同时服务多个客户（租户）。这是很普通的 SaaS（软件服务化）的解决方案。在这些系统中隔离不同用户的信息（数据、定制内容，等等）是一个很大的挑战。这包括每个租户的数据保存在数据库中。最后这一部分，有时也称为 multi-tenant data（多租户数据）正是我们关注的内容。

## 16.2. 多租户数据处理方法

在多租户系统中主要有 3 种方法隔离信息，这些方法是伴随着不同数据库 schema 的定义与 JDBC 设置的。

注意

每一种方法都有优点与缺点，各自的具体技术与注意事项。这些主题超出了本文档的范围。深入探讨这些主题的资源大量存在。比如 http://msdn.microsoft.com/en-us/library/aa479086.aspx 中就有大量的内容涉及到这些主题。

### 16.2.1. Separate databa（独立数据库）

每个租户数据保持在物理的独立的数据库实例中。JDBC 连接指向这些具体的每个数据库，一些连接池服务于每个租户。通常应用程序做法是为每个租户定义连接池，通过租户的“tenant identifier（租户标识符）”选择连接池，这些标识符与当前登录的用户相关联。

### 16.2.2. 独立的 schema

每个租户的数据保存在一个数据库实例的不同 schema 中。这有两种不同的方法定义 JDBC 连接：

连接指向每个具体的 schema,与独立数据库方法一样。为连接提供以下两个选项：1、驱动支持通过连接 URL，连接默认的 schema；2、连接池机制支持命名 schema。我们要为每个租户定义连接池，通过租户的 “tenant identifier（租户标识符）”选择连接池，这个标识符与当前登录的用户相关联。

连接指向数据库本身（使用默认的 schema），但是连接的 SQL 要更改为 SET SCHEMA（或相类似的）指令。使用此方法，我们要为所有的租户提供一个连接池，但是在使用连接之前，一定要将引用的 schema 命名为“tenant identifier（租户标识符）”，这个标识符与当前登录的用户相关联。

### 16.2.3. 数据分区（鉴别器）

所有数据保存在一个数据库 schema 中，每个租户的数据通过分区值或鉴别器进行划分。鉴别器的复杂度是从简单的列值到复杂的 SQL 公式。此方法为所有的租户提供一个连接池。但是应用此方法的程序必须更改每条 SQL 语句，为每条 SQL 语句加上“tenant identifier（租户标识符）”作为鉴别器。

## 16.3. Hibernate 中的 Multi-tenancy（多租户）

使用多租户的 Hibernate 时归结为两点：一是 API，二是集成片段。Hibernate 一直努力保持 API 的简单并且与底层复杂的整合隔离开。API 其实传递特定租户标识符打开的会话，并且做为会话的一部分内容。

**示例 16.1. 从 `SessionFactory` 指定租户标识符**

```
Session session = sessionFactory.withOptions()
                        .tenantIdentifier( yourTenantIdentifier )
                        ...
                        .openSession();
```

另外，设定策略是，使用 hibernate.multiTenancy 指定 org.hibernate.MultiTenancyStrategy 名（译者注：策略名）。Hibernate 将按你指定的策略类型执行验证。策略就是上面讨论的隔离方法。

NONE

（默认）不期望使用多租户。事实上，如果使用这个策略，当租户标识符传给 session 的 open 方法时会产生错误。

SCHEMA

使用独立的 schema 方法。使用此策略，如果打开 session 时没有传入租户标识符会产生错误。另外，必须指定 MultiTenantConnectionProvider。

DATABASE

使用独立的数据库方法。使用此策略，如果打开 session 时没有传入租户标识符会产生错误。另外，必须指定 MultiTenantConnectionProvider。

DISCRIMINATOR

使用数据分区（鉴别器）方法。使用此策略，如果打开 session 时没有传入租户标识符会产生错误。此策略不能在 Hibernate 4.0 或 4.1 中实现。计划支持 5.0。（译者注：其它版本这里没说，估计只能用于 5.0）

### 16.3.1. MultiTenantConnectionProvider

当使用 DATABASE、SCHEMA 策略时，Hibernate 需要能够得到指定租户的连接方法。MultiTenantConnectionProvider 约定就是做这个的。应用程序开发者需要提供这个约定的实现。它们大多数方法是自解释的。唯一不是自解释的方法是 getAnyConnection 与 releaseAnyConnection。这很重要，注意这些方法不接受租户标识符。Hibernate 使用这些方法在启动期间执行各种设置，这主要是通过 java.sql.DatabaseMetaData 对象。

通过指定以下方法来使用 MultiTenantConnectionProvider：

使用 hibernate.multi_tenant_connection_provider 设置。它能够命名 MultiTenantConnectionProvider 实例，此实例是 MultiTenantConnectionProvider 实现类的引用或者 MultiTenantConnectionProvider 实现类的名称。

将它直接传入 org.hibernate.boot.registry.StandardServiceRegistryBuilder。

如果上面的选项没有匹配，但是设置指定了 hibernate.connection.datasource 的值，Hibernate 将假设它要使用具体的 DataSourceBasedMultiTenantConnectionProviderImpl 实现类，当程序运行在一个应用程序服务器中，并且使用每租户一个 javax.sql.DataSource 时，这是一个非常合理的假设。更多细节参考 javadoc。

### 16.3.2. CurrentTenantIdentifierResolver

org.hibernate.context.spi.CurrentTenantIdentifierResolver 是一个约定，这个约定可以让 Hibernate 解析什么是当前租户的标识符。它的实现通过这个类的 setCurrentTenantIdentifierResolver 方法是直接指定的。它也可以通过 hibernate.tenant_identifier_resolver 进行设定。

当使用 CurrentTenantIdentifierResolver 时有两种情况：

第一种情况是当应用程序使用 org.hibernate.context.spi.CurrentSessionContext 结合多租户时。用到 current-session 时，如果在当前范围内找不到 session,Hibernate 将要打开一个 Session。但是在多租户环境下打开 session 时租户标识符是必须指定的。这时就用到了 CurrentTenantIdentifierResolver；Hibernate 将询问你提供的实现决定如何使用租户标识符打开 Session。在这种情况下，CurrentTenantIdentifierResolver 必须提供。

另一种情况是当你从不显式指定租户标识符时，就像在示例 16.1,“从 SessionFactory 指定租户标识符”中一样。如果 CurrentTenantIdentifierResolver 已经指定，当打开 session 时，Hibernate 将使用它来决定默认标识符是什么。

另外，如果 CurrentTenantIdentifierResolver 实现类的 validateExistingCurrentSessions 方法返回 true。Hibernate 将确保当前范围的所有 session 都有相匹配的租户标识符。这种能力只适用于在当前会话的设置中使用了 CurrentTenantIdentifierResolver 时。

### 16.3.3. 缓冲

Hibernate 中的多租户无缝支持 Hibernate 二级缓冲，缓冲的 key 就是由用户标识符编码而来。

### 16.3.4. 杂项

当前的 schema 的出口不会真正的与多租户一起工作。这可能不会改变。
JPA 专家组正在改进即将到来的 2.1 版，增加多租户支持的定义。

## 16.4. `MultiTenantConnectionProvider` 实现策略

**示例 16.2. 用不同的连接池实现**

```
        /**
         * 简单实现了两个连接池，利用{@link
org.hibernate.service.jdbc.connections.spi.AbstractMultiTenantConnect
ionProvider}类的支持
         */
        public class MultiTenantConnectionProviderImpl extends
AbstractMultiTenantConnectionProvider {
                private final ConnectionProvider acmeProvider =
ConnectionProviderUtils.buildConnectionProvider( "acme" );
                private final ConnectionProvider jbossProvider =
ConnectionProviderUtils.buildConnectionProvider( "jboss" );

                @Override
                protected ConnectionProvider
getAnyConnectionProvider() {
                        return acmeProvider;
                }

                @Override
                protected ConnectionProvider
selectConnectionProvider(String tenantIdentifier) {
                        if ( "acme".equals( tenantIdentifier ) ) {
                                return acmeProvider;
                        }
                        else if ( "jboss".equals( tenantIdentifier ) )
{
                                return jbossProvider;
                        }
```

```
                        throw new HibernateException( "Unknown tenant
identifier" );
                }
        }
```

上面的方法是有效的数据库方法。也是有效的 Schema 方法，这个 schema 由底层数据库通过 connection  URL 连接提供，并且是可以命名的。

**示例 16.3. 使用单一连接池的 MultiTenantConnectionProvider 实现**

```
        /**
         * 简单实现单个连接池，利用"connection altering"，使用多
schemas 服务。这里我们使用 T-SQL 的 USE 指令；
         * Oracle 用户可以使用 ALTER SESSION SET SCHEMA 指令
         */
        public class MultiTenantConnectionProviderImpl
                        implements MultiTenantConnectionProvider,
Stoppable {
                private final ConnectionProvider connectionProvider =
ConnectionProviderUtils.buildConnectionProvider( "master" );

                @Override
                public Connection getAnyConnection() throws
SQLException {
                        return connectionProvider.getConnection();
                }

                @Override
                public void releaseAnyConnection(Connection
connection) throws SQLException {

        connectionProvider.closeConnection( connection );
                }

                @Override
                public Connection getConnection(String
tenantIdentifier) throws SQLException {
                        final Connection connection =
getAnyConnection();
                        try {
```

```
        connection.createStatement().execute( "USE " +
tenanantIdentifier );
                        }
                        catch ( SQLException e ) {
                                throw new HibernateException(
                                                "Could not alter JDBC
connection to specified schema [" +

        tenantIdentifier + "]",
                                                e
                                );
                        }
                        return connection;
                }

                @Override
                public void releaseConnection(String tenantIdentifier,
Connection connection) throws SQLException {
                        try {

        connection.createStatement().execute( "USE master" );
                        }
                        catch ( SQLException e ) {
                                // on error, throw an exception to
make sure the connection is not returned to the pool.
                                // your requirements may differ
                                throw new HibernateException(
                                                "Could not alter JDBC
connection to specified schema [" +

        tenantIdentifier + "]",
                                                e
                                );
                        }

        connectionProvider.closeConnection( connection );
                }

                ...
        }
```

这种方法只能涉及到 Schema 方法。

# 第 17 章 OSGi

## 概要

Open Services Gateway initiative(OSGi)（开放服务网关协议）规范描述一个动态、模块化的系统。“Bundles”（组件）可以在运行时安装、激活、停用、卸载，不需要重启系统。OSGI 框架管理 bundles 的依赖、包、类。框架也负责管理 bundles 之间的类加载，包的可见性。此外，服务的注册与发现由“whiteboard（白板）”模式 提供。

OSGi 环境目前存在大量的特殊的挑战。主要是运行期可用 bundles 的动态特性，可能需要大量的体系结构上的考虑。同时架构必须允许 OSGi 特有的类加载与服务的注册、发现。

**目录**

## 17.1. OSGi 规范与环境

Hibernate 针对 OSGi4.3 或更高版本。从 4.3 开始是必须的，不要用 4.2，因为 Hibernate 依靠 OSGi 的 BundleWiring 进行实体/映射的扫描。（译者注：4.2 中没有这个功能）

Hibernate 支持三种类型的 OSGi 配置

1. *容器管理的 JPA*：17.5, “容器管理的 JPA”
2. *非托管的 JPA*: 17.6, “非托管的 JPA ”
3. *非托管 Native*：17.7, “非托管 Native ”

## 17.2. hibernate-osgi

并没有将 OSGi 功能嵌入 hibernate-core, hibernate-entitymanager, sub-modules, 所以 hibernate-osgi 出现了。这是有目的的分离，隔离所有的 OSGi 依赖。这里提供 OSGi 特有的 ClassLoader（类加载器）（聚合了容器的 CL 和实体管理器的 CL）（译者注：CL=ClassLoader），JPA 持久化提供者，SF/EMF 启动程序，实体/映射扫描器，服务管理。

## 17.3. features.xml

Apache Karaf 环境趋向于大量使用它的“features”的概念，这里的 feature 是一套 order-specific bundle，这些 bundle 聚焦在如何简化操作。这些 feature 通常定义在 features.xml 文件中。Hibernate 创建与发布了自己的 features.xml，里面定义了核心的 hibernate-orm,以及额外的附加的可选的 feature（缓冲，Enver，等等）。这包括二进制分发，以及已经部署好的 JBoss Nexus 资源库（使用 org.hibernate 的 groupID 和 hibernate-osgi，里面还包括带有 classifier 的 karaf.xml 文件 ）
注意我们的 features 使用与封装的 ORM 组件相同的版本。也要注意我们做的大量测试是针对 Karaf3.0.3,这做为我们基于 PaxExam 的集成测试的一部分。然而，它们有可能在其它版本下也可以工作。
理论上，hibernate-osgi 支持各种 OSGi 容器，比如 Equinox。在此情况下，请参考 features.xml，了解哪些 bundle 是必需激活的，它们正确的顺序是什么。然而，注意，Karaf 启动后大量 bundle 是自动加载的。其中可能有要手工加载的。

## 17.4. 快速入门与演示

所有的三种配置都在 QuickStart/Demo（快速入门与演示）中，请查看 hibernate-demos 工程:

## 17.5. 容器管理的 JPA

企业级 OSGi 规范包括管理 JPA 的容器。这个容器负责在 bundles 中发现持久层单元,并自动创建 EntityManagerFactory（每 PU 一个 EMF）。这里使用的 JPA 提供者就是 hibernate-osgi,它已经自己注册成了一个 OSGiPersistenceProvider 服务。

### 17.5.1. 企业级 OSGi 的 JPA 容器

为了使用容器管理的 JPA，一个企业级 OSGi 的 JPA 容器必须在运行时被激活。在 Karaf 中，这意味着 Aries JPA，Aries JPA 是即插即用的（简单的激活 jpa 与 transaction feature）。原本，我们打算在我们的 features.xml 中包括这些依赖。但是经过 Karaf 与 Aries 团队的指导后，将它们拉出来了（译者

注：放在外面，用户自己设置）。这保证了 HibernateOSGi 的可移植性，不用直接关联到 Aries 的版本，而让用户自己选择使用什么。
话虽如此，QuickStart/Demo（快速入门/演示）项目中包括一个 features.xml 的样本，这个样本展示了在 Karaf 中为了支持这个环境哪些 features 需要激活。正如提到的，使用这个纯粹就是一个参考。

### 17.5.2. persistence.xml

类似其它的 JPA 设置，你的 bundle 必须包括一个 persistence.xml 文件。它通常位于 META-INF。

### 17.5.3. DataSource（数据源）

典型的企业级 OSGi JAP 用法是在容器中包括一个安装好的数据源。你 bundle 的 persistence.xml 文件通过 JNDI 调用这个数据源。示例中，你可以安装下面的 H2 DS。你可以手工部署 DS（Karaf 有一个 deploy 目录（译者注：文件夹）），或者通过"blueprint bundle"（blueprint:file:/[PATH]/datasource-h2.xml））。

**示例 17.1. datasource-h2.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
        <!--
        First install the H2 driver using:
        > install -s mvn:com.h2database/h2/1.3.163

        Then copy this file to the deploy folder
        -->
        <blueprint
xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

          <bean id="dataSource" class="org.h2.jdbcx.JdbcDataSource">
                <property name="URL"
value="jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE"/>
                <property name="user" value="sa"/>
                <property name="password" value=""/>
          </bean>
```

```
        <service interface="javax.sql.DataSource" ref="dataSource">
             <service-properties>
               <entry key="osgi.jndi.service.name"
value="jdbc/h2ds"/>
             </service-properties>
        </service>
      </blueprint>
```

然后通过你的 persistence.xml 的 persistence-unit（持久化单元）使用 DS（数据源）。下面是 Karaf 的设置，如果是其它的容器，名称可能要变化一下。

**示例 17.2. META-INF/persistence.xml**

```
<jta-data-
source>osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc
/h2ds)</jta-data-source>
```

#### 17.5.4. Bundle 包的导入

你 bundle 的 manifest 至少导入：
javax.persistence
org.hibernate.proxy 与 javassist.util.proxy, 由于 Hibernate 能够返回一个懒加载的代理（javassist 强化发生在实体类加载器的运行时）

#### 17.5.5. 获取 EntityManger（实体管理器）

很简便，支持得很好，获得 EntityManager 方法是利用 OSGi bundle 中的 OSGI-INF/blueprint/blueprint.xml 文件。容器使用你持久化单元的名称（译者注：下例中的 unitname 的值）自动注入 EntityManager 的实例到你给的 bean 属性中。

**示例 17.3. OSGI-INF/blueprint/blueprint.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
        <blueprint  default-activation="eager"

        xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"

        xmlns:jpa="http://aries.apache.org/xmlns/jpa/v1.0.0"

        xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.0.0">

          <!-- This gets the container-managed EntityManager and
injects it into the DataPointServiceImpl bean.
          Assumes DataPointServiceImpl has an "entityManager" field
with a getter and setter. -->
          <bean id="dpService"
class="org.hibernate.osgitest.DataPointServiceImpl">
                  <jpa:context unitname="managed-jpa"
property="entityManager"/>
                  <tx:transaction method="*" value="Required"/>
          </bean>
          <service ref="dpService"
interface="org.hibernate.osgitest.DataPointService" />
        </blueprint>
```

## 17.6. 非托管 JPA

Hibernate 也支持通过 hibernate-entitymanager 使用 JPA，这个 JPA 不是由 OSGi 容器管理。客户的 bundle 负责管理 EntityManagerFactory（实体管理器工厂）与 EntityManagers（实体管理器）。

### 17.6.1. persistence.xml

类似其它的 JPA 设置，你的 bundle 必须包括一个 persistence.xml 文件。它通常位于 META-INF。

### 17.6.2. Bundle 包的导入

你 bundle 的 manifest 至少导入：

javax.persistence
org.hibernate.proxy 与 javassist.util.proxy，由于 Hibernate 能够返回一个懒加载的代理（ javassist 强化发生在实体类加载器的运行时）
JDBC 驱动包(示例:org.h2)
org.osgi.framework，需要发现 EMF（译者注：EntityManagerFactory）（下面描述）

### 17.6.3. 获取 EntityMangerFactory

hibernate-osgi 使用 JPA 的 PersistenceProvider 接口的全名注册了一个 OSGi 服务。然后引导程序为 OSGi 环境建立一个指定的 EntityManagerFactory。下面内容很重要，你的 EMF 通过服务获得而不是手工建立的。服务控制着 OSGi 的 ClassLoader，发现扩展点，扫描等等。手工建立的 EntityManagerFactory 在运行时无法工作。

**示例 17.4. 发现/使用 EntityManagerFactory**

```
public class HibernateUtil {

		private EntityManagerFactory emf;

		public EntityManager getEntityManager() {
			return
getEntityManagerFactory().createEntityManager();
		}

		private EntityManagerFactory getEntityManagerFactory()
{
			if ( emf == null ) {
				Bundle thisBundle =
FrameworkUtil.getBundle( HibernateUtil.class );
				BundleContext context =
thisBundle.getBundleContext();

				ServiceReference serviceReference =
context.getServiceReference( PersistenceProvider.class.getName() );
```

```
                                PersistenceProvider
persistenceProvider = (PersistenceProvider)
context.getService( serviceReference );

                                emf =
persistenceProvider.createEntityManagerFactory( "YourPersistenceUnitN
ame", null );
                        }
                        return emf;
                }
        }
```

## 17.7. 非托管 Native

Navtive(原生)Hibernate 也是支持的。客户 bundle 负责管理 SessionFactory 与 Sessions。

### 17.7.1. Bundle 包的导入

你 bundle 的 manifest 至少导入：
javax.persistence
org.hibernate.proxy 与 javassist.util.proxy，由于 Hibernate 能够返回一个懒加载的代理（javassist 强化发生在实体类加载器的运行时）
JDBC 驱动包(示例: org.h2)
org.osgi.framework，必须发现 SF（译者注：SessionFactory）（下面描述）
org.hibernate.*包，必须（如：cfg,criterion,服务等等.）

### 17.7.2. 获取 SessionFactory

hibernate-osgi 使用 SessionFactory 接口的全名注册一个 OSGi 服务，引导程序为具体的 OSGi 环境建立 SessionFactory 的实例。下面内容很重要：你的 SF（译者注：SessionFactory）通过 OSGi 环境获得，服务控制着 OSGi ClassLoader，发现扩展点，扫描等等。手工建立的 SessionFactory 在运行时无法工作。

**示例 17.5. 发现/使用 SessionFactory**

```
public class HibernateUtil {

                private SessionFactory sf;

                public Session getSession() {
                        return getSessionFactory().openSession();
                }

                private SessionFactory getSessionFactory() {
                        if ( sf == null ) {
                                Bundle thisBundle =
FrameworkUtil.getBundle( HibernateUtil.class );
                                BundleContext context =
thisBundle.getBundleContext();

                                ServiceReference sr =
context.getServiceReference( SessionFactory.class.getName() );
                                sf = (SessionFactory)
context.getService( sr );
                        }
                        return sf;
                }
        }
```

## 17.8. 可选模块

非托管的 navtive 演示工程展示了使用可选 Hibernate 模块。每个模块添加额外依赖的 bundle 时一定要先激活，不管是手工的还是额外 feature 的。ORM4.2 完全支持 Envers。对于 C3P0，Proxool，EhCache，Infinispan 的支持加入到 4.3 中了，但是这些第三方的类库都无法在 OSGi 下工作（主要是 ClassLoader 的问题，等等）。我们会在 JIRA 中跟踪这些问题。

## 17.9. 扩展点

多个约定都允许应用程序整合并扩展 hibernate 的功能。很多应用利用 JDK 服务来实现，hibernate-osgi 同样支持它们在 OSGi 服务中。三种配置方式都可

以实现并注册它们。hibernate-osgi 在 EMF/SF 引导时，将发现并整合它们。支持的扩展点如下所示。当服务注册时必须使用指定的接口。

- org.hibernate.integrator.spi.Integrator(4.2)
- org.hibernate.boot.registry.selector.StrategyRegistrationProvider (4.3)
- org.hibernate.boot.model.TypeContributor(4.3)
- JTA 的 javax.transaction.TransactionManager

javax.transaction.UserTransaction(4.2), 但是这此通常是由 OSGi 容器提供的。

最简单的实现扩展点的方法就是通过 blueprint.xml 文件。添加 OSGI-INF/blueprint/blueprint.xml 到你的 classpath(类路径)。Envers 的 blueprint 就是一个很好的例子：

**示例 17.6. 在 blueprint.xml 中注册扩展点**

```
<!--
          ~ Hibernate, Relational Persistence for Idiomatic Java
          ~
          ~ License: GNU Lesser General Public License (LGPL),
version 2.1 or later.
          ~ See the lgpl.txt file in the root directory or
<http://www.gnu.org/licenses/lgpl-2.1.html>.
          -->
        <blueprint default-activation="eager"

xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

                <bean id="integrator"
class="org.hibernate.envers.boot.internal.EnversIntegrator" />
                <service ref="integrator"
interface="org.hibernate.integrator.spi.Integrator" />

                <bean id="typeContributor"

        class="org.hibernate.envers.boot.internal.TypeContributorImpl
" />
                <service ref="typeContributor"
interface="org.hibernate.boot.model.TypeContributor" />
```

```
        </blueprint>
```

扩展点也可以用编程的方式进行注册。注册使用 BundleContext#registerService 方法，这一过程通常是在 BundleActivator#start 方法中进行。

## 17.10. 附加说明

从技术上来讲，多个持久化单元是支持企业 OSGi JAP 与非托管的 Hibernate JPA。但是，当前不支持它们使用在 OSGi 中。在 Hibernate4 时，只有一个 OSGi 特定的 ClassLoader 实例，它被每个 Hibernate bundle 使用，主要是因为大量使用了静态的 TCCL 应用（译者注：容易出现误导，作者可能想说整合 OSGi 中只有一个 ClassLoader）。我们希望在 Hibernate5 中支持每个持久化单元一个 OSGi ClassLoader（译者注：那到底现在 5 中支持否？这可是 5 的文档！）。

扫描支持发现非显式列出的实例与映射。但是它们必须在同一个持久化单元中的 bundle。我们的 OSGi ClassLoader 只顾及“请求的 bundle”（所以必需由服务建立 EMF/SF），而不是试图扫描所有可用的 bundle。这主要是为了版本控制与冲突保护等。

一些容器（如：Aries）中，PersistenceUnitInfo#excludeUnlistedClasses 的返回值一直是 true，即使你在 persistence.xml 中显式指定 exclude-unlisted-classes 为 false 也无效。它们声称这是保护 JPA 提供者（“我们为你处理一切”）。即使我们还是在许多情况下支持它们。解决的办法是设置 hibernate.archive.autodetection 的值（译者注：这是你想自动扫描的内容），例如：hbm,class，这是告诉 Hibernate 忽视 excludeUnlistedClasses 的值，扫描*.hbm.xml 与实体。

扫描目前不支持在 package-info.java 中注释的包。

目前，Hibernate OSGi 主要的测试在 Apache Karaf 与 Apache Aries 中完成。其它的容器需要进一步测试如：Equinox，Gemini，其它容器提供者。

Hibernate ORM 有许多依赖，目前不提供 OSGi mainfests。

QuickStart tutorials（快速入门教程）中使用了大量第三方的 bundle(SpringSource, ServiceMix)或 wrap:...操作。

# 第 18 章 Envers

**概要**

Hibernate Envers 的目的是为你应用程序的实体数据提供历史版本。非常像源代码控制管理工具，如：Subversion 或 Git，Hibernate Envers 管理一个 revisions（译者注：版本、修订、校对）的概念，这个管理的前提是你的应用程序数据使用了 audit table。每个事务关联到一个全局的 revision number（版本号、修订号），此版本号可以用于标识组的更改（非常类似代码控制器中的变更设置）。这个 revisions（版本、修订、校对）是全局的，有一个版本号，你就可以查询这个版本中的各种实体，检索这个版本中数据库（部分）视图。你可以从日期中找到版本，反过来讲，你可以从已经提交的版本中得到日期。

**目录**

## 18.1. 基础知识

audit（译者注：直译成审计，感觉审查更合适，有判断与检查的意思,所以不译）一个实体的更改过程，你只要做 2 件事： hibernate-envers 加入 classpath;@Audited 注释加入到实体定义。

重要的

不同于以前的版本，你不需要在 Hibernate 配置文件中指定监听器。只要将 Envers jar 包放到 classpath 就足够了--监听器将自动注册。

与原来一样，你可以创建、修改、删除实体。如果你看一下实体生成的 schema，或是已经被 Hibernate 持久化的数据，你将发现与原来没有任何变化。然而，对于每个被 audit 的实体，引入了一个新的表-entity_table_AUD（译者注：audit 表默认是在原表后加上_AUD），每当你提交事务时，这里就存储了历史数据。Envers 自动国建立 Audit（审查）表，前提是你将 hibernate.hbm2ddl.auto 选项设定为 create，create-drop 或 update。另外，想以编程的方式输出完整的数据库 schema，使用

org.hibernate.envers.tools.hbm2ddl.EnversSchemaGenerator。也可以创建合适的 DDL 语句，有关的 Ant 任务本手册稍后介绍。
代替注释整个类，Audit 所有的属性，你可以使用@Audited 只注释一些要持久化的属性。这样将只对这些属性进行 audit。
对于实体的 audit（历史记录），你可以使用 AuditReader 接口进行访问。它通过 AuditReaderFactory 的方法传入打开的 EntityManager 或 Session 获得。（译者注：AuditReader ar=AuditReaderFactory.get(session)）参考 javadoc 了解这些类所提供功能的细节。

## 18.2. 配置

可以配置 Hibernate Envers 各方面的行为，如表名等等。

**表 18.1. Envers 配置属性**

| 属性名 | 默认值 |
| --- | --- |
| org.hibernate.envers.audit_table_prefix | |
| org.hibernate.envers.audit_table_suffix | _AUD |
| org.hibernate.envers.revision_field_name | REV |
| org.hibernate.envers.revision_type_field_name | REVTYPE |
| org.hibernate.envers.revision_on_collection_change | true |
| org.hibernate.envers.do_not_audit_optimistic_locking_field | true |

| | |
|---|---|
| org.hibernate.envers.store_data_at_delete | false |
| org.hibernate.envers.default_schema | null（与被a |
| org.hibernate.envers.default_catalog | null（与被a |
| org.hibernate.envers.audit_strategy | org.hiberna |
| org.hibernate.envers.audit_strategy_validity_end_rev_field_name | REVEND |
| org.hibernate.envers.audit_strategy_validity_store_revend_timestamp | false |

| | |
|---|---|
| org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name | REVEND_TSTM |
| org.hibernate.envers.use_revision_entity_with_native_id | true |
| org.hibernate.envers.track_entities_changed_in_revision | false |
| org.hibernate.envers.global_with_modified_flag | false,可以单<br>写 |
| org.hibernate.envers.modified_flag_suffix | _MOD |

| | |
|---|---|
| org.hibernate.envers.embeddable_set_ordinal_field_name | SETORDINAL |
| org.hibernate.envers.cascade_delete_revision | false |
| org.hibernate.envers.allow_identifier_reuse | false |

重要的

以下的配置选项是最近加上的，应视为测试：

1. org.hibernate.envers.track_entities_changed_in_revision
2. org.hibernate.envers.using_modified_flag
3. org.hibernate.envers.modified_flag_suffix

## 18.3. 额外的映射注释

audit 表的名称可以在实体中设置，使用@AuditTable 注释。为每个被 audit 实体添加这个注释可能很烦人，所以有可能最好还是使用前缀/后缀。

如果你有 secondary tables（第二个表）的映射，它们的 audit 表用同样的方式创建（通过前缀与后缀）。如果你希望重写这种行为，你可以使用@SecondaryAuditTable 与@SecondaryAuditTables 注释。

如果你希望重写一些字段/属性的 audit 行为，你可以继承@Mappedsuperclass 或一个嵌入的组件。你可以使用@AuditOverride(s)注释在子类或是组件上。

如果你想 audit 关系映射，这些关系包括@OneToMany+@JoinColumn。请参考 18.11,"映射异常"，那里有@AuditJoinTable 的描述，是你想要看的。

如果你想 audit 一个关系，目标实体没有进行 audit（例如字典类的实体，不会更改所以不用 audit），只需要用 @Audited(targetAuditMode =

RelationTargetAuditMode.NOT_AUDITED)注释。然后，当读取实体的版本历史记录时，这个关系一直关联到“当前”的关联对象。如果关联对象在数据库是不存在，Envers 默认行为是抛出 javax.persistence.EntityNotFoundException 异常。应用@NotFound(action = NotFoundAction.IGNORE)注释消除异常并且为关系赋值为 null。注意：此解决方案导致 to-one(对一)关系的隐式立即加载。

如果你想 audit 父类实体的属性，父类中没有显式 audit(没有用@Audited 注释整个类与任何属性)，你可以在子类@Audited 注释的 auditParents 属性中注列出父类。请注意 auditParents 属性已经过时了，请使用 @AuditOverride(forClass = SomeEntity.class, isAudited = true/false)代替。

## 18.4. 选择 audit 策略

基本配置后就是重要的选择 audit 策略，audit 策略用于持久化和检索 audit 信息。这是一种在持久化性能与查询性能之间的权衡，当前有两种 audit 策略。默认的 audit 策略开始 revision(校对，修订)同时持久 audit 数据。被 audit 实体的每一行添加、更新、删除都会在 audit 表中加入一行或多行数据,这与有效的 revision(校对，修订)是同时开始的。audit 表的数据添加后就不会被修改。可以从 audit 表中使用子查询查询适当的 audit 信息。这些子查询非常慢而且索引困难。

另一种是 validity  audit 的策略。这种策略保存 audit 信息的开始版本与结束版本。被审查实体的每一行添加、更新、删除都会在 audit 表中加入一行或多行数据，这与有效的 revision(校对，修订)是同时开始的。但是，同时前面那些行的 end-revision（最终版本字段）被赋值成这个版本（如果有这个字段）。查询 audit 信息时可以使用“开始版本与结束版本”代替默认策略中的子查询。

这种策略的结果是持久化有点慢，因为涉及到额外的更新操作。但是检索 audit 信息很快。这可以通过添加额外的索引来改善。

## 18.5. 版本日志

**版本日志的数据**

当 Envers 开始一个新的版本时，它建立一个新的 *revision entity（版本实体）*，实体中保存着版本信息。默认的信息至少包括：

*revision number（版本号）*--一个整数值（int/Integer 或者 long/Long），是版本的主键。

*revision timestamp（版本时间戳）*-或是 long/Long 或是 java.util.Date，它的值代表建立版本的时间。注意：当使用 java.util.Date 代替 long/Long 作为时间戳时，不要将它存储于会丢失精度的列上（译者注：列的类型建议到毫秒）。

Envers 用一个实体处理这些信息。默认情况下使用一个内部类表示这个实体，并将实体映射到 REVINFO 表。然而，你可以用你自己的方法收集额外的细节，比如谁更改了实体，请求 IP 地址来自哪里。想实现这些工作你要做 2 件事：

首先，你需要告诉 Envers 你希望使用哪个实体。你的实体必须使用 @org.hibernate.envers.RevisionEntity 注释。这个实体必须注释 2 个属性，分别为@org.hibernate.envers.RevisionNumber 与 @org.hibernate.envers.RevisionTimestamp，为了继承所有需要的行为，你也可以继承 org.hibernate.envers.DefaultRevisionEntity 类。

上面操作可以简单的将一个自然类添加为自定义的版本实体。Envers 将“发现它”。注意，如果有多个实体标记@org.hibernate.envers.RevisionEntity 注释，将产生错误。

其次，你需要告诉 Envers 如何建立你自定义版本类的实体，这通过 org.jboss.envers.RevisionListener 接口的 newRevision 的方法完成。

你通过@org.hibernate.envers.RevisionEntity 注释，告诉 Evers 使用自定义 org.hibernate.envers.RevisionListener 的实现。如果从你自定义的 @RevisionEntity 类无法访问 RevisionListener（如：不在同一个模块中），可以使用 org.hibernate.envers.revision_listener 实现类的全名作为注释属性的值。还有一种指定监听器的方法，是通过配置参数重写版本实体的 value 属性，一定是全类名。

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

        public class MyCustomRevisionListener implements
RevisionListener {
```

```
                public void newRevision(Object revisionEntity) {
                        ( (MyCustomRevisionEntity) revisionEntity )...;
                }
        }
```

另一种替代使用 org.hibernate.envers.RevisionListener 的方法是：调用 org.hibernate.envers.AuditReader 接口的 getCurrentRevision 方法得到当前的版本对象，然后向版本对象中填充信息。这个方法接收 persist 参数，这个参数代表是否版本实体在返回前被持久化。true 确保不管 audit 实体是否有变化，返回的有访问标识符（版本号、主键）。false 意味着版本号为 null，但如果 audit 实体有变化，版本实体就会持久化。

**示例 18.1. 在版本对象中保存用户名**

ExampleRevEntity.java

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

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

        import javax.persistence.Entity;

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

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

ExampleListener.java

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

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

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


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

#### 18.5.1. 在版本控制中跟踪修改的实体名

默认情况下，不跟踪每个版本中实体变化类型（译者注：增、删、改）。这暗示为了检索指定版本的变化需要查询所有存储 audit 数据的表。Envers 提供了一种简单的机制，就是建立 REVCHANGES 表，此表用于存储已变化的持久化对象的实体名（译者注：默认为类名）。每条记录封装了版本的标识符（REVINFO 表的外键）和一个字符串。

跟踪修改的实体名可以使用下面 3 种不同的方法：

设置 org.hibernate.envers.track_entities_changed_in_revision 参数为 true。在此情况下，隐式使用 org.hibernate.envers.DefaultTrackingModifiedEntitiesRevisionEntity 作为版本日志实体。

建立一个自定义的版本实体，这个类继承自 org.hibernate.envers.DefaultTrackingModifiedEntitiesRevisionEntity 类

```
1. @Entity
2.         @RevisionEntity
3.         public class ExtendedRevisionEntity
4.                                 extends
   DefaultTrackingModifiedEntitiesRevisionEntity {
5.                 ...
        }
```

在版本实体类上自定义一个字段，此字段用 @org.hibernate.envers.ModifiedEntityNames 注释。这个属性必须是 Set<String>类型

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

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

                        ...
        }
```

用户使用上面方法之一，就可以检索指定版本的所有实体变化。参考 18.7.4,"按版本查询实体的修改"中 API 的使用。

用户还可以自定义跟踪机制，这个机制用于跟踪实体变化。在此情况下，建立 org.hibernate.envers.EntityTrackingRevisionListener 接口的实现类，此类当成@org.hibernate.envers.RevisionEntity 注释的值。EntityTrackingRevisionListener 公开一个方法（译者注：entityChanged（）方法），此方法在当前版本范围内，接收被 audit 实体添加、修改、删除时发布的通知。

**示例 18.2. 自定义跟踪各个版本中实体变化的实现类**

CustomEntityTrackingRevisionListener.java

```
        public class CustomEntityTrackingRevisionListener
                                implements
EntityTrackingRevisionListener {
                @Override
                public void entityChanged(Class entityClass, String
entityName,

Serializable entityId, RevisionType revisionType,
```

```
                                                                    Object
revisionEntity) {
                        String type = entityClass.getName();

        ((CustomTrackingRevisionEntity)revisionEntity).addModifiedEnt
ityType(type);
                }

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

CustomTrackingRevisionEntity.java

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

                @RevisionTimestamp
                private long customTimestamp;

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


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

                ...
        }
```

ModifiedEntityTypeEntity.java

```
        @Entity
        public class ModifiedEntityTypeEntity {
```

```
                @Id
                @GeneratedValue
                private Integer id;

                @ManyToOne
                private CustomTrackingRevisionEntity revision;

                private String entityClassName;

                ...
        }

CustomTrackingRevisionEntity revEntity =

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

## 18.6. 在属性级别上跟踪实体变化

默认情况下，Envers 版本信息中只保存更改后的实体。这个方法让我们可以建立实体属性值的历史更改记录并对它进行 audit 查询。当我们不只关注结果值的变化，也关注类型的变化时，这也可以保存每一个版本的额外 metadata 信息并让我们使用。在 18.5.1,"在版本控制中跟踪修改的实体名"中所描述的功能中，可以告诉我们在指定版本中哪些实体更改了，这里我们更进一步，“更改标识”可以让 Envers 跟踪指定的版本中哪些属性更改了。

跟踪实体属性级别的变化可以通过：

设置 org.hibernate.envers.global_with_modified_flag 为 true。这个全局设置导致所有被 audit 实体的所有属性被加上“更改标志”。（译者注：跟踪所有被 audit 实体的全部属性。）

在属性或实体定义时，使用@Audited(withModifiedFlag=true)注释。

权衡此功能从两个方面，1 是 audit 表增大了，2 是 audit 写过程性能下降（很小，可以忽略）。这是由于实际上每个被跟踪的属性都有一个伴随的 boolean 列，这个列在 schema 判断是否跟踪属性更改。当然，这是 Envers 的工作，不需要开发者做额外的工作。由于涉及到性能，当想使用 granular configuration（粒度配置）方式时，推荐使用这一功能。

如何使用“Modified Flags（更改标志）”，看看这些简单的查询 API 如何使用，参考 18.7.3,“通过指定属性的变化查询实体的版本”。

## 18.7. 查询

你可以从两个维度来查看历史数据。第一-水平-给定版本的数据库状态。这样，你就可以查询处于第 N 个版本的实体。第二-垂直-就是各版本中实体的变化。因此，你们可以给出实体的变化来查询版本。
Envers 的查询与 Hibernate Criteria 查询相似，所以如果你常用 Criteria 查询，那么使用 Envers 查询很简单。
当前查询实现的主要限制是你不能使用关系。你只能在关系实体的 id 上指定约束，并且只能在“主”的一侧（译者注：主从关系主的一方）。这可以会在未来版本中改进。
请注意，这些针对 audit 数据的查询大多数情况下很慢，这是与对应的“实时”数据相比，因为它们都涉及到相关的子查询。
未来, 当使用 valid-time audit 策略时，也就是当保存实体的开始版本和结束版本时，查询将在两个方面有所改进，一是速度，二是可选择性。参考 18.2,“配置”。

### 18.7.1. 指定一个版本查询类的实体

这类查询的切入点是：

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

然后你可以指定约束，满足约束条件的实体将返回。添加约束可以使用 AuditEntity 工厂类的相应方法完成。示例中：只选择“name”属性等于“John”的实体。

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

选择与指定实体相关的实体：

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
        //或
```

```
        query.add(AuditEntity.relatedId("address").eq(relatedEntityId)
);
```

你可以用习惯的方式，限制结果的数量，排序结果，使用聚合与 projections（译者注：映射，select 的内容）（除了分组）。当你的查询完成后，你可以调用 getSingleResult()或 getResultList() 方法得到结果。

一个完整的查询如下：

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

### 18.7.2. 按实体类的变化，查询版本

这类查询的切入点是:

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

与上面的一样，你可以为这个查询添加约束。这里有一些额外的可能性：

使用 AuditEntity.revisionNumber()，你可以指定约束，projections（译者注：映射，select 的内容），排序，查询被 audit(审查)实体的修改。

同样，使用 AuditEntity.revisionProperty(propertyName)，你可以对版本实体的属性指定约束，projections（译者注：映射，select 的内容），排序，查询相关被 audit 实体的修改。

AuditEntity.revisionType()让你可以访问上面的版本类型（ADD, MOD, DEL）（译者注：添加、修改、删除）

使用这些方法，你可以按版本号进行排序，设置 projection，约束版本号大小或小于指定值等等，如示例中：下面的例子中查询最小的版本号，条件是：MyEntity 类的实体；id=entityId;版本号大于 42：

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

```
                .add(AuditEntity.revisionNumber().gt(42))
                .getSingleResult();
```

你在查询中可以使用的第二种额外的特性是：查询最大的版本或最小的版本的能力。例如：如果你想查询一个版本号，实体的 actualDate 属性大于指定值，但是版本号尽可能的小：

```
Number revision = (Number) getAuditReader().createQuery()
                .forRevisionsOfEntity(MyEntity.class, false, true)
                // 我们只对第一个版本有兴趣
                .setProjection(AuditEntity.revisionNumber().min())
                .add(AuditEntity.property("actualDate").minimize()

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

minimize()与 maximize()方法返回一个 criteria，你可以继续在这个 criteria 上加约束，但是所用到实体的属性一定是可以比较大小的。

AggregatedAuditExpression#computeAggregationInInstanceContext()用于分别计算每个实体实例的上下文中聚合表达式的值。当查询所有实体指定类型的最新版本时很有用。

你可能注意到了，有两个 boolean 的对数传递给的建立查询的方法。第一个 boolean 参数是 selectEntitiesOnly，它只在不显式指定 projection 时有效（译者注：如果用了 setProjection 方法，此参数无效），如果为 true，查询的结果是一个实体列表（满足约束中指定的版本变化）。

如果为 false,结果是三列组成的一个列表。第一列是实体实例。第二列是版本对象（如果没有自定义版，就是 DefaultRevisionEntity 的实例）。第三列是版本类型（RevisionType 枚举之一：ADD, MOD, DEL）。

第二个 boolean 参数是 selectDeletedEntities，是指结果中是否包含已经删除的实体。如果为 ture,结果中包含版本类型为 DEL 的所有实体（除了 ID，其它属性为 null 的）。

### 18.7.3. 通过指定属性的变化查询实体的版本

对于上面的描述的两种类型的查询也可以用指定的 Audit criteria 调用 hasChanged()与 hasNotChanged()来实现，完成 18.6,"在属性级别上跟踪实体

变化"所描述的功能。它们最适用于垂直查询，但是现在的 API 并不限制它们使用在水平查询中。让我们看下面的例子：

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

这个查询返回给定 ID 的 MyEntity 的所有版本，其中这些 MyEntity 的 actualDate 属性已经变化了。使用这个查询不会返回与 actualDate 无关的其它版本。当然，没有什么可以阻止用户将 hasChanged 条件与 criteria 加上方法组合使用，这是一种很正常的方法。

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

这个查询返回 MyEntity 在指定的版本号的水平切片。这些返回的结果限制在 prop1 属性已经修改，但 prop2 属性没有修改。注意结果集通常只包含版本号小于指定的版本号，因为我们不能将这个查询看做“给我所有的 MyEntitie 变化，指定版本号的 prop1 属性已经修改，但是 prop2 属性没有涉及”。想得到这些结果，我不得不使用 forEntitiesModifiedAtRevision 查询：

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

#### 18.7.4. 按版本查询实体的修改

基本的查询允许检索指定版本的实体名与相应 java 类型的变化。

```
Set<Pair<String, Class>> modifiedEntityTypes = getAuditReader()

        .getCrossTypeRevisionChangesReader().findEntityTypes(revision
Number);
```

其它查询（也是从 org.hibernate.envers.CrossTypeRevisionChangesReader 可以访问的）：

*List<Object> findEntities(Number)*--返回指定版本的被 audit 实体的变化（添加，更新，删除）快照。执行 n+1sql 查询，其中 n 是指定版本的不同实体 class 更改的数量。

*List<Object> findEntities(Number, RevisionType)*返回指定版本的被 audit 实体的变化（添加，更新，删除）快照，第二个参数是指定修改的类型。执行 n+1sql 查询，其中 n 是指定版本的不同实体 class 更改的数量。

*Map<RevisionType, List<Object>> findEntitiesGroupByRevisionType(Number)*--返回一个 map 包含按修改类型（如：添加，更新，删除）分组的实体列表。执行 3n+1sql 查询，其中 n 是指定版本的不同实体 class 更改的数量。

注意：上面描述的方法可以合法的使用默认机制跟踪实体名称的变化（参考 18.5.1,"在版本控制中跟踪实体的名称"）。

## 18.8. 条件 audit

Envers 持久化 audit 数据是各种 Hibernate 事件的回应（如更新后，插入后，等等）。使用来自 org.hibernate.envers.event.spi 包的一套监听器。默认情况下，如果 Envers 的 jar 包加入到 classpath，这些监听器由 hibernate 自动注册。

条件 audit 可以重写这些 Envers 事件监听器。使用自定义的 Envers 事件监听，下面是所需的步骤：

设置 Hibernate 属性 hibernate.listeners.envers.autoRegister 为 false,关闭自动 Envers 事件监听器注册。

建立合适的事件监听器的子类。如果你想有条件的 audit 实体的插入操作，继承 org.hibernate.envers.event.spi.EnversPostInsertEventListenerImpl 类。将条件 audit 的业务逻辑放置到子类中，然后调用父类的方法完成 audit 的执行。

建立你自己的 org.hibernate.integrator.spi.Integrator 的实现，就像 org.hibernate.envers.boot.internal.EnversIntegrator 一样。使用你自定义的事件监听器替换默认的实现（译者注：integrator 合成器）。

想在 Hibernate 启动时自动启用 integrator（合成器），你需要向你的 jar 包中添加 META-INF/services/org.hibernate.integrator.spi.Integrator 文件。此文件中应包含接口实现类的全名。

## 18.9. 理解 Envers Schema

每个被 audit 的实体（即实体至少包含一个被 audit 的字段），都将建立一个 audit 表。默认情况下，audit 表的名称在原始表名后加上"_AUD"后缀，但是这可以由两种方式重写，1、在配置中指定默认的后缀与前缀；2、在实体定义时使用@org.hibernate.envers.AuditTable 注释。

### audit 表的列

1. 原始实体的 ID（在联合主键时可以多于一列）
2. 版本号--一个整数值。与版本实体表中的版本号相对应。
3. 版本类型-短整
4. 原始实体的被 audit 字段。

audit 表的主键是原始对象 ID 与版本号的组合（译者注：就是上面 ID 与版本号两列组成的联合主键）--它代表一个给定实体实例在给定版本上的历史记录。当前的实体数据同时存储在原始表与 audit 表中，这是一种数据的重复，但是这种解决案可以提高查询系统效率同时减少内存的使用，希望这不会给用户带来不方便。audit 表的一行包含：ID-实体的 id;N-版本；D-数据，这也就意味着：从版本 N 开始 id 为 ID 的实体，数据是 D。因此，如果你想发现在版本 M 的实体，你不得不在 audit 表中查找版本号小于或等于 M 的行，但是很可能找不到，如果没有找到这样的行或找到的行有“deleted”标记，这就意味着实体在这个版本中不存在了。

"revision  type（版本类型）"字段目前可以是 3 个值：0，1，2，分别表示 ADD（添加），MOD（修改）与 DEL（删除）。一行数据如果版本类型为 DEL，只有一个实体的 ID，没有其它数据（所有字段 为 NULL），就是说“这个实体已经在这个版本中删除了”。

另外，有一个 revision entity table（版本实体表），这个表包含全局的版本信息。默认情况下，表名为 REVINFO，只包含 2 列：ID 与 TIMESTAMP。每次有了新版本就会向此表中加入一行。也就是：每一次有关 audit 数据变化

的事件的提交（译者注：新版本是指原始数据的变化）。这个表的名称可以配置，行的名称、附加的行也是一样（译者注：也可以配置）。这些详述在 18.5,"版本日志"中。

虽然全局版本是一种很好的方式，它提供了正确的 audit 关系，但一些人提出这可能是系统的瓶颈，因为这些数据经常更改。一种有效的解决方案是引入实体的“locally revisioned（本地版本）”的选项，这个选项将独立创建。这个方法不能正确的反应版本关系，但是也不需要 REVINFO 表。另一种可能的解决方案是引入“revisioning groups（版本组）”的概念：这个组中的实体共享版本号。每个组由以下内容组成：一或多个紧密连接的组件，这些组件组成一个关系图，关系图代表了实体之间的关系。在论坛中，你对于这些主题的建议是很受欢迎的！：）

## 18.10. 使用 Ant 生成 schema

如果你喜欢使用 Hibernate Tools Ant task 生成数据库 schema。你可能注意到文件中不包含 audit 表的定义。想生成 audit 表，你只需要简单的用 org.hibernate.tool.ant.EnversHibernateToolTask 替换 org.hibernate.tool.ant.HibernateToolTask 就要可以了。前面的类是后面的类的扩展，只是增加了建立版本实体的内容。现在你可以像原来一样使用 Ant task 了。

示例:

```
<target name="schemaexport" depends="build-demo"
          description="Exports a generated schema to DB and file">
          <taskdef name="hibernatetool"

        classname="org.hibernate.tool.ant.EnversHibernateToolTask"
                classpathref="build.demo.classpath"/>

          <hibernatetool destdir=".">
                <classpath>
                  <fileset refid="lib.hibernate" />
                  <path location="${build.demo.dir}" />
                  <path location="${build.main.dir}" />
                </classpath>
                <jpaconfiguration persistenceunit="ConsolePU" />
                <hbm2ddl
                  drop="false"
```

```
                create="true"
                export="false"
                outputfilename="versioning-ddl.sql"
                delimiter=";"
                format="true"/>
        </hibernatetool>
    </target>
```

将生成下面的 schema:

```
    create table Address (
                        id integer generated by default as identity
(start with 1),
                        flatNumber integer,
                        houseNumber integer,
                        streetName varchar(255),
                        primary key (id)
                );

                create table Address_AUD (
                        id integer not null,
                        REV integer not null,
                        flatNumber integer,
                        houseNumber integer,
                        streetName varchar(255),
                        REVTYPE tinyint,
                        primary key (id, REV)
                );

                create table Person (
                        id integer generated by default as identity
(start with 1),
                        name varchar(255),
                        surname varchar(255),
                        address_id integer,
                        primary key (id)
                );

                create table Person_AUD (
                        id integer not null,
                        REV integer not null,
                        name varchar(255),
                        surname varchar(255),
                        REVTYPE tinyint,
                        address_id integer,
```

```
                        primary key (id, REV)
                );

                create table REVINFO (
                        REV integer generated by default as identity
(start with 1),
                        REVTSTMP bigint,
                        primary key (REV)
                );

                alter table Person
                        add constraint FK8E488775E4C3EA63
                        foreign key (address_id)
                        references Address;
```

## 18.11. 映射异常

### 18.11.1. 现在不会将来也不会支持的

Bags，它能包含 non-unique（译者注：非唯一、重复、相同）的元素。想象一下 bags 的持久操作。例如一个字符串的 bag，违反关系数据库的原则：每个表是 tuple(元组)的 set(集)（译者注：不要理解为 collection，因为 E 文 set 与 collection 不同，但中文一样）。当使用 bags 的情况下（也可能是连接表），如果有两个重复的元素，这两个元组对应的同一个元素。当试图持久化两个一样的元素时，Hiberante 允许，但是 Envers（准确说是数据库连接器）将抛出异常。原因就是违反唯一性原则。
如果你一定要用 bag,至少有两种方法：
通过@IndexColumn 注释，使用一个索引集合。
通过@CollectionId 注释，为你的元素提供一个唯一标识。

### 18.11.2. 现在不会将来会支持的

使用@CollectionId 注释为 bag 集合定义一个标识列(JIRA ticket HHH-3950)。

### 18.11.3. @OneToMany+@JoinColumn

当集合映射同时有这两种注释，Hibernate 不会产生 join table(连接表)。然而 Envers 会产生。所以当你读取关联实体已经变化的版本时，你不会出现失败的结果。
为额外的 join table（连接表）命名可以用特定的注释：@AuditJoinTable，与 JPA 的 @JoinTable 语法一样。
一种很特殊的情况是：在关系映射同时使用@OneToMany+@JoinColumn 在一边，@ManyToOne+@JoinColumn(insertable=false, updatable=false)在另一边。这种关系其实是双向的，但是只在 owning(译者注：是指“一”那侧，也就是主端)边有集合。
为了正确使用 Envers audit 这些关系，你可以使用@AuditMappedBy 注释。它让你可以指定 reverse property（译者注：反转属性）（使用 mappedBy 元素）。在索引集合的情况下，索引列必须在引用实体中映射（使用 @Column(insertable=false, updatable=false)），并且具体指定 positionMappedBy。这个注释只影响 Envers 的工作。注意 ，这个注释是试验性的，未来可以会变化。

## 18.12. 高级：Audit 表分区

### 18.12.1. audit 表分区的好处

因为 audit 表趋向无穷大，同时可能快速增长。当 audit 表增长受到一定的限制（来源于各种 RDBMS 并且/或者 操作系统，且不相同），合理的处理方式是使用表分区。SQL 表分区提供了很多优越性，但一定不限制：
通过有选择的在各个分区间移动行（或者清理旧行）来提高查询性能。
快速加载数据，建立索引等等。

### 18.12.2. 选择合适的列为 audit 表分区

一般 SQL 表按表中存在的列进行分区。作为一个规范，趋向于使用 *end revision（最终版本）*或者 *end revision timestamp（最终版本时间戳）*列为 audit 表分区。
注意
最终版本信息在默认 audit 策略中不可用。

因此下面的 Envers 配置选项是必须的:

```
org.hibernate.envers.audit_strategy=org.hibernate.envers.strategy.ValidityAuditStrategy
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp=true
```

或者，你也可以使用下面属性重写默认值：

```
org.hibernate.envers.audit_strategy_validity_end_rev_field_name
```

更多信息参考 18.2,“配置”。
使用版本号对 audit 表进行分区的原因是基于一种假设，假设 audit 表应该按“increasing level of interestingness(成长关注度的级别)”进行分区，好像：
有些 audit 表分区的数据是不（或不长久）受关注的。它们可以保存在速度慢一些介质中，甚至可能最终被清除。
有些 audit 表的分区数据是被可能关注的。
某些 audit 表的分区数据最可能受到关注。它们应该保存在读写最快的介质中。

### 18.12.3. Audit 表分区示例

为了确认哪列会成为 increasing level of interestingness(成长关注度的级别)。考虑一个简单的例子，这个例子是某机构工资登记表。
目前，工资表包含下面的行，代表某人 X 的工资：

**表 18.2. 工资表**

| Year(年份) | Salary（工资）(USD) |
|---|---|
| 2006 | 3300 |
| 2007 | 3500 |
| 2008 | 4000 |

| | |
|---|---|
| 2009 | 4500 |

本财年（2010）的工资是未知的。机构要求一个财年所有登记的工资的变体都要被记录（即 audit 跟踪）。在特定日期做出后台决策是基于当时登记的工资。在任何时间必须可以再现原因（译者注：多次决策结果相同），为何当时做出这样的决策。
下面是 audit 的可用信息，是按发生顺序保存的。

**表 18.3. 工资-audit 表**

| Year(年份) | Revision type（版本类型） | Revision timestamp（版本时间戳） | Salary（工资）(USD) | End revision timestamp（结束版本时间戳） |
|---|---|---|---|---|
| 2006 | ADD | 2007-04-01 | 3300 | null |
| 2007 | ADD | 2008-04-01 | 35 | 2008-04-02 |
| 2007 | MOD | 2008-04-02 | 3500 | null |
| 2008 | ADD | 2009-04-01 | 3700 | 2009-07-01 |
| 2008 | MOD | 2009-07-01 | 4100 | 2010-02-01 |
| 2008 | MOD | 2010-02-01 | 4000 | null |
| 2009 | ADD | 2010-04-01 | 4500 | null |

#### 18.12.3.1. 确定合适的分区列

为了分区数据，“关注度级别”必须定义。考虑下面内容：
对于 2006 财年只有一个版本。它是 audit 表中所有行 *revision timestamp（版本时间戳）* 最老的数据，但是它仍应该视为受关注的，因为工资表中 2006 财年数据中 *end revision timestamp（最终版本时间戳）* 是 NULL。
还要注意在 2011 年更新 2006 财年的工资是很少发生的（但是有可能，至少在一个财年后的 10 年中），audit 信息应该移到慢介质中（基于 *revision timestamp* 的长短）。记住，在这种情况下，Envers 将更新最近修改行的 *end revision timestamp* 列。
2007 财年在工资表中有两个版本，它们有几乎相同的 *revision timestamp（版本时间戳）* 列，但是 *end revision timestamp* 列不同。一看就知道，明

显第一个版本是错误的，不用关注。2007 财年唯一需要关注的版本是 *end revision timestamp（最终版本时间戳）*列为 NULL 的那行。
基于上面的说，明显只有 *end revision timestamp* 列适合 audit 表分区。*revision timestamp* 列不合适。

#### 18.12.3.2. 确定合适的分区方案

工资表可能的分区方案如下：
*end revision timestamp* year = 2008
这个 audit 表分区的内容不用关注或不用长时间关注。
*end revision timestamp* year = 2009
这个 audit 表分区的内容可能关注。
*end revision timestamp* year >=2010 or null
这个 audit 表分区的内容非常关注
这种分区方案也涉及到一个潜在的问题，如果 audit 表某行被修改时，同时也要更新 *end revision timestamp* 列。即使 Envers 更新 *end revision timestamp* 列为系统时间是在瞬间完成的，也要将这些行保存在同一分区中（“extension bucket”）（译者注：比如 2009 年数据更新后，要在表中加一个版本同时修改 2009 的老版本，所以 2009 年的数据要在一个分区中）。
2011 年的某一时刻，最后一个分区（或’extension bucket’）被分割成 2 个新分区：
*end revision timestamp* year = 2010
这个 audit 表分区包含可能关注的（在 2011 年）。
*end revision timestamp* year >= 2011 or null
这个 audit 表分区包含最关注的，是一个新的’extension bucket’。

## 18.13. Envers 有关的链接

Hibernate 主页
Envers 论坛
JIRA 问题跟踪（当发布有关 Envers 有关的问题，要选择“Envers”板块）
IRC 频道
Envers 博客
FAQ

# 第 19 章. 数据库可移植性思考

**目录**



## 19.1. 可移植性基础

Hibernate 的卖点之一（真正的整体对象/关系映射）就是数据库可移植性的观念。这可能意味着一个 IT 内部用户从一个数据库迁移到另一个，或者也可能意味着一个框架或应用使用 hibernate 时同时针对多个数据用户产品。不管怎样，基本想法就是 Hibernate 帮助你遇到多个数据库时不用更改原代码，理论上不用更改映射 metadata.

## 19.2. Dialect(方言)

可移植性第一件事就是 Dialect（方言），这是由 org.hibernate.dialect.Dialect 约定来限定的。方言封装了 Hibernate 与不同数据库的通信，协同完成工作如：得到 sequence（序列）值或构建 SELECT 查询。Hibernate 捆绑了广泛的主流数据库的各种方言。如果你发现你特定的数据为没有包含在内，你可以写自己的方言，这不很困难。

## 19.3. Dialect（方言）解析

最初，Hibernate 需要用户指定使用什么 Dialect（方言）。在这种情况下，用户同时使用多个数据库是有问题。通常这需要这些用户配置 Hibernate  Dialect（方言）或定义它们自己的方法设定相关值。

从 3.2 版本开始，Hibernate 引入了自动检测 Dialect（方言）的概念，可以从与数据库连接的 java.sql.Connection 中得到 java.sql.DatabaseMetaData。这是一个改进，希望这解决了数据库的限制。Hibernate 可以提前知道 Dialect（方言）是什么，不用再配置或重写 Dialect（方言）了。
从 3.3 版本开始，Hibernate 引入了更强大的方法自动确定使用的 Dialect（方言），这是依靠一系统的代理，这些代理实现了 org.hibernate.dialect.resolver.DialectResolver 类，这个类中有一个唯一的方法：

```
public Dialect resolveDialect(DatabaseMetaData metaData) throws
JDBCConnectionException
```

这里基本约定是如果解析器“理解”数据库 metadata，它将返回对应的 Dialect（方言）；如果不理解就返回 null，然后由下一个解析器继续处理。从方法签名可以看出，有可能抛出 org.hibernate.exception.JDBCConnectionException 异常。JDBCConnectionException 异常理解为一个“非临时的”（不可恢复）的连接问题出现，立即停止解析。其它异常结果发出一个警告然后继续到下一个解析器。
最酷的部分是用户可以自己注册自定义的解析器，这些自定义的解析器在 Hibernat 内建的解析器之前。在下面这些情况下这是很有用的：这允许方便的集成 Hibernate 自动检测中不包含的方言；这允许你指定其它公开数据库的自定义方言；等等。如果想注册一或多个解析器，简单地（用逗号，制表符，空格分隔）使用’hibernate.dialect_resolvers 配置设定。（参考 org.hibernate.cfg.Environment 中的 DIALECT_RESOLVERS 常量）

## 19.4. 标识符生成

当考虑在数据库间移植时，另一个重要的确认是选择你想使用的标识符生成策略。起初，Hibernate 为此提供 *native* 生成器，这个生成器是在 *sequence*, *identity*, *table* 之间选择，当然这要依赖于底层数据库的能力。然而这有一个潜在的问题，就是一些数据库支持 *identity* 生成器，一些数据库不支持。*identity* 生成器依赖 SQL 中定义一个 IDENTITY(或　自增)列来管理标识符的值；这就是我们知道的 post-inser（插入后）主键（标识符）生成策略，因为在我们知道主键值前，insert 必须实际发生。因为 Hibernate 依赖这个标

识符在持久化上下文中唯一引用实体，所以不管当前事务语义如何，都会立即发生 insert。

注意

Hibernate 发生了一点改变。使用之更好理解，以便在可能的情况下 insert 被延迟执行。

问题的根本在于，应用程序自身，如果遇到上面的问题，最好是应用程序语义改进一下。

从 3.2.3 版本开始，Hibernate 内置了一套 enhanced（增强）标识符生成器，为数据库的可移植性提供了一种完成不同的方式解决上面的问题。

注意

这里介绍 2 种绑定 *enhanced（增强）*生成器的方法：

org.hibernate.id.enhanced.SequenceStyleGenerator
org.hibernate.id.enhanced.TableGenerator

这些生成器背后的想法是：为不同的数据库提供一个实际的切入点，以方便生成标识符。例如：org.hibernate.id.enhanced.SequenceStyleGenerator 模仿数据库 sequence 行为，如果数据库不支持 sequence，就使用 table。

## 19.5. 数据库函数

警告

在这个领域 Hibernate 需要改进。涉及到在移植性方面，这些函数从 HQL 中处理当前的工作很好； 然而其它方面相当不足。

用户可以用很多方式引用 SQL 函数。然而，不是所有的数据库都支持相同的函数。Hibernate 提供了一种方法使用函数，映射函数的 *logical* 名称到一个代理，这个代理知道如何使用指定的函数，甚至用完全不同的物理函数调用。

重要的

技术上来说，这些函数通过 org.hibernate.dialect.function.SQLFunctionRegistry 类进行注册，这个类允许用户提供自定义函数定义，但是不包括自定义方言。这些特殊的行为还没有完全完成（译者注：功能还在开发中）。

用户可以使用 org.hibernate.cfg.Configuration 以编程的方式完成函数注册，这些函数可以被 HQL 识别。

## 19.6. 类型映射

本节稍后完成……
可能需要的内容： jpa 可移植性，HQL/JPQL 的不同，命名策略，基本类型，简单 ID 类型，生成的 ID 类型，组合列与 many-to-one，插件的 ID

# 附录 A Legacy Bootstrapping（过时的引导方式）

**目录**



legacy（过时、旧版）的引导方式 SessionFactory 是通过 org.hibernate.cfg.Configuration 对象。从本质上讲，配置表现为一个切入点，这个点指定了所有建立 SessionFactory 的内容：一切从配置中来，如映射，如策略等等。我们可以想象 Configuration(配置对象)就像一个大锅一样，我们加入了很多原料（映射，设置等等），最终得到了 SessionFactory。
注意

这个方法有一些重大的缺点，导致旧版本过时，发展出了新的方法。这些在 3.1, "Native 引导"已经讨论过了。Configuration 是半过时的，但是还可能使用。在限制形式后，消除了 Configuration 的缺点。本手册一开始，Configuration 使用了新的引导代码，使用自动发现机制自动寻找各种信息。

你可以直接实例化得到 Configuration。你可以指定映射 metadata(xml 映射文件，注释类)，这些描述了你的应用程序的对象模型与 SQL 数据库之间的映射关系。

**示例 A.1. Configuration 用法**

```
Configuration cfg = new Configuration()
                                // addResource does a classpath
resource lookup
                                .addResource("Item.hbm.xml")
                                .addResource("Bid.hbm.xml")
```

```
                                // calls addResource using
"/org/hibernate/auction/User.hbm.xml"

        .addClass(org.hibernate.auction.User.class)

                                // parses Address class for mapping
annotations
                                .addAnnotatedClass( Address.class )

                                // reads package-level (package-
info.class) annotations in the named package
                                .addPackage( "org.hibernate.auction" )

                                .setProperty("hibernate.dialect",
"org.hibernate.dialect.H2Dialect")

        .setProperty("hibernate.connection.datasource",
"java:comp/env/jdbc/test")
                                .setProperty("hibernate.order_updates",
"true");
```

其它指定配置信息的方法，包括：
放置 hibernate.properties 文件到 classpath 的根目录
传递 java.util.Properties 的实例到 Configuration#setProperties
通过 Hibernate cfg.xml 文件
系统属性中使用 java '-Dproperty=value'

## A.1. 迁移

映射配置的方法对应新 API 中相应的方法……

### 映射 metadata

Configuration#addFile

MetadataSources#addFile
Configuration#add(XmlDocument)

No replacement
Configuration#addXML

No replacement
Configuration#addCacheableFile

MetadataSources#addCacheableFile
Configuration#addURL

MetadataSources#addURL
Configuration#addInputStream

MetadataSources#addInputStream
Configuration#addResource

MetadataSources#addResource
Configuration#addClass

MetadataSources#addClass
Configuration#addAnnotatedClass

MetadataSources#addAnnotatedClass
Configuration#addPackage

MetadataSources#addPackage
Configuration#addJar

MetadataSources#addJar
Configuration#addDirectory

MetadataSources#addDirectory
Configuration#registerTypeContributor

MetadataBuilder#applyTypes
Configuration#registerTypeOverride

MetadataBuilder#applyBasicType

## 设置

Configuration#setProperty

StandardServiceRegistryBuilder#applySetting
Configuration#setProperties

No replacement
Configuration#addProperties

StandardServiceRegistryBuilder#applySettings
Configuration#setNamingStrategy

No replacement. NamingStrategy split into implicit/physical strategies
Configuration#setImplicitNamingStrategy

MetadataBuilder#setImplicitNamingStrategy
Configuration#setPhysicalNamingStrategy

MetadataBuilder#setPhysicalNamingStrategy
Configuration#configure

StandardServiceRegistryBuilder#configure
Configuration#setInterceptor

SessionFactoryBuilder#applyInterceptor
Configuration#setEntityNotFoundDelegate

SessionFactoryBuilder#applyEntityNotFoundDelegate
Configuration#setSessionFactoryObserver

SessionFactoryBuilder#addSessionFactoryObservers
Configuration#setCurrentTenantIdentifierResolver

SessionFactoryBuilder#applyCurrentTenantIdentifierResolver

## 附录 B Legacy（过时的）Hibernate Criteria 查询

**目录**



注意

附录中的涵盖了旧版本的 org.hibernate.CriteriaAPI，这些是过时的，新版本关注点是 JPA 中的 javax.persistence.criteria.CriteriaQueryAPI。最终 Hibernate 指定的 criteria 迁移到 JPA 的 javax.persistence.criteria.CriteriaQuery。更多 JPA API 的细节参数 第 14 章,*Criteria*

这此信息从老的 Hibernate 文档复制。

Hibernate 的特点是直观的、可扩展的 criteria 查询 API。

## B.1. 建立 Criteria 实例

接口 org.hibernate.Criteria 表示一个特定持久化类的查询。Session 是 Criteria 实例的工厂。

```
Criteria crit = sess.createCriteria(Cat.class);
        crit.setMaxResults(50);
        List cats = crit.list();
```

## B.2. 减小结果集

一个独立的 criterion 查询是 org.hibernate.criterion.Criterion 接口的实例 org.hibernate.criterion.Restrictions 类定义了一个工厂方法得到内置的某一 Criterion 的类型。

```
List cats = sess.createCriteria(Cat.class)
                .add( Restrictions.like("name", "Fritz%") )
                .add( Restrictions.between("weight", minWeight,
maxWeight) )
                .list();
```

Restrictions（约束）可以进行逻辑分组。

```
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();
```

这里有一系列内置 criterion 类型（Restrictions 的子类）。最大的用处是允许你直接指定 SQL。

```
List cats = sess.createCriteria(Cat.class)
                .add( Restrictions.sqlRestriction("lower({alias}.name)
like lower(?)", "Fritz%", Hibernate.STRING) )
                .list();
```

{alias}是一个点位符，这个占位符替换查询实体行的别名。

你可以从 Property（属性）实例中得到 criterion。你可以通过调用 Property.forName()，建立一个 Property：

```
Property age = Property.forName("age");
        List cats = sess.createCriteria(Cat.class)
                .add( Restrictions.disjunction()
                        .add( age.isNull() )
                        .add( age.eq( new Integer(0) ) )
                        .add( age.eq( new Integer(1) ) )
                        .add( age.eq( new Integer(2) ) )
                ) )
                .add( Property.forName("name").in( new String[]
{ "Fritz", "Izi", "Pk" } ) )
                .list();
```

## B.3. 排序结果集

你可以使用 org.hibernate.criterion.Order 对结果集进行排序

```
List cats = sess.createCriteria(Cat.class)
                .add( Restrictions.like("name", "F%")

        .addOrder( Order.asc("name").nulls(NullPrecedence.LAST) )
```

```
                .addOrder( Order.desc("age") )
                .setMaxResults(50)
                .list();
List cats = sess.createCriteria(Cat.class)
                .add( Property.forName("name").like("F%") )
                .addOrder( Property.forName("name").asc() )
                .addOrder( Property.forName("age").desc() )
                .setMaxResults(50)
                .list();
```

## B.4. 关联

通过导航关联，使用 createCriteria()，你可以在关联实体上指定约束：

```
List cats = sess.createCriteria(Cat.class)
                .add( Restrictions.like("name", "F%") )
                .createCriteria("kittens")
                        .add( Restrictions.like("name", "F%") )
                .list();
```

第二个 createCriteria()返回一个新的 Criteria 实例，这个实例将查询出 kittens 的集合。

这是一种替代形式，在某些情况下更有用。

```
        List cats = sess.createCriteria(Cat.class)
                .createAlias("kittens", "kt")
                .createAlias("mate", "mt")
                .add( Restrictions.eqProperty("kt.name", "mt.name") )
                .list();
```

（createAlias()不是建立一个新的 Criteria 的实例。）

前两个查询返回的 Cat 实例，持有 kittens 集合，这些有 kittens 集合没有被 criteria 预先过滤。如果你想要得到与 criteria 匹配的 kittens，你需要使用 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");
        }
```

另外你可以用 left outer join（左外连接）处理结果集：

```
        List cats = session.createCriteria( Cat.class )
                                                .createAlias("mate",
"mt", Criteria.LEFT_JOIN, Restrictions.like("mt.name", "good%") )

            .addOrder(Order.asc("mt.age"))
                                                .list();
```

这将返回的内容是：有配偶的 Cat，并且配偶名字以"good"开始+所有没有配偶的 Cat，并且按配偶的年龄排序（译者注：因为是 out join ，所以包含配偶为 NULL 的数据）。当处于下面两种情况时这很有用：1、当需要排序或者限制数据库预先返回的复杂/大量的结果集时，2、删除内存中多个查询执行过程中产生的结果集实例时。
由于没有这个功能，第一个查询所有没有配偶的 cat 要再另外的查询中加载。
第二个查询将检索所有配偶名以"good"开始的猫，并按配偶年龄排序。
第三个查询，内存中的 list 需要手工连接起来。

## B.5. 动态关联抓取

你可以通过 setFetchMode()在运行时动态定义关联的抓取模式。

```
        List cats = sess.createCriteria(Cat.class)
                .add( Restrictions.like("name", "Fritz%") )
                .setFetchMode("mate", FetchMode.EAGER)
                .setFetchMode("kittens", FetchMode.EAGER)
                .list();
```

这个查询将使用 outer join 抓取 mate(配偶)与 kittens（小猫）

## B.6. Components(组件)

针对一个可嵌入式的 component（组件）属性加约束，当建立 Restriction（约束）时，component（组件）属性的名称要预先指定到约束的属性上。criteria 对象应该建立自己的实体，不能由 component（组件）自己建立。例如，Cat 有一个 component（组件）属性 fullName，这个 fullName 属性有两个子属性 firstName 与 lastName：

```
        List cats = session.createCriteria(Cat.class)

        .add(Restrictions.eq("fullName.lastName", "Cattington"))
                                .list();
```

注意：这个 component（组件）属性不能用于集合查询，参考下面 B.7,"集合"。

## B.7. 集合

当使用 criteria 用于集合时，有两种不同的情况。第一种是集体中包含实体（如<one-to-many/>或<many-to-many/>）或 component（组件）（<composite-element/> ），第二种是集合中包含 scalar（标量）值（<element/>）。第一种情况的语法在上面 B.4,"关联"介绍过了，在例子中我们约束了 kittens 集合。本质上，我们针对集合属性建立 Criteria 对象，并用这个实例约束实体或 component（组件）属性。
对于查询只有标量值的集合（译者注：上面的第二种情况），我们仍然针对集合建立 Criteria。但是如果有引用值，我们可以使用特殊属性"elements"。如果是一个索引集合，我们也可以索引属性，这是通过"indices"属性指定的。

```
        List cats = session.createCriteria(Cat.class)
                                .createCriteria("nickNames")
                                        .add(Restrictions.eq("elements",
"BadBoy"))
                                .list();
```

## B.8. Example 查询

org.hibernate.criterion.Example 类，允许以给出的实例做模板，构建一个 criterion 查询。

```
        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，标识符与关联被忽略。默认情况下，不包含 null 值属性。

你可以自行调节 Example 如何应用。

```
        Example example = Example.create(cat)
                .excludeZeroes()           //排除零值属性 exclude
zero valued properties
                .excludeProperty("color")  //排除属性名称为"color"的
属性 exclude the property named "color"
                .ignoreCase()              //字符串比较时忽视大小写
perform case insensitive string comparisons
                .enableLike();             //字符串比较时可以使用 like
操作符 use like for string comparisons
        List results = session.createCriteria(Cat.class)
                .add(example)
                .list();
```

你甚至可以在关联对象上使用 example

```
        List results = session.createCriteria(Cat.class)
                .add( Example.create(cat) )
                .createCriteria("mate")
                        .add( Example.create( cat.getMate() ) )
                .list();
```

## B.9. Projections（投影），聚合与分组

org.hibernate.criterion.Projections 类是 Projection（投影）实例的工厂。

你可以在查询中调用 setProjection()来使用 projection。

```
        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();
```

在 criteria 查询中不需要显示指定“group by”。当然，projection 类型中也定义了 *grouping projections(分组投影)*，这些分组投影最终出现在 SQL 的 group by 子句中。

可以给 projection 分配一个别名，这个别名可以用于约束或排序。这里有两种不同的方式实现别名：

```
        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();
```

alias()andas()方法可以简单的封装 projection 实例的别名。做为一种快捷方式，当你添加向 projection 列表中加入 projection 时，你可以为 projection 分配别名。

```
        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();
```

你也可以使用 Property.forName()表示一个别名。

```
        List results = session.createCriteria(Cat.class)
                .setProjection( Property.forName("name") )
                .add( Property.forName("color").eq(Color.BLACK) )
                .list();

        List results = session.createCriteria(Cat.class)
                .setProjection( Projections.projectionList()

        .add( Projections.rowCount().as("catCountByColor") )

        .add( Property.forName("weight").avg().as("avgWeight") )

        .add( Property.forName("weight").max().as("maxWeight") )

        .add( Property.forName("color").group().as("color" )
                )
                .addOrder( Order.desc("catCountByColor") )
                .addOrder( Order.desc("avgWeight") )
                .list();
```

## B.10. Detached queries（分离式查询）与子查询

DetachedCriteria 类允许你在 Session 的作用域外建立查询，并使用任意的 Session 执行这些查询。

```
        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();
```

DetachedCriteria 也可以表示子查询。如果想得到一个子查询的 Criteria，也以通过 Subqueries 或者 Property 的相关方法取得。

```
        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();
```

关联子查询也可能是：

```
        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();
```

基于子查询的多列约束示例：

```
        DetachedCriteria sizeQuery =
DetachedCriteria.forClass( Man.class )
```

```
        .setProjection( Projections.projectionList().add( Projections.
property( "weight" ) )

                .add( Restrictions.eq( "name", "John" ) );
        session.createCriteria( Woman.class )
                .add( Subqueries.propertiesEq( new String[]
{ "weight", "height" }, sizeQuery ) )
                .list();
```

## B.11. 通过 natural（自然）ID 查询

大多数查询，包括 criteria 查询，这些查询的效率很低，这是因为查询缓冲失效的情况经常发生（译者注：缓冲点中率低）。然而，你可以使用一种特别的查询来优化缓冲算法：通过常量 natural key（自然主键）进行查询。在很多应用程序中，这种形式的查询经常发生。Criteria API 提供这种专用查询所需的一切。

首先使用<natural-id>映射你实体的 natural key（自然主键），然后确保可以使用二级缓冲。

```
        <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>
```

此功能不适用于带有 *mutable（可变）* natural keys（自然主键）的实体。

一旦你启动了 Hibernate 查询缓冲，Restrictions.naturalId()允许你使用更有效的缓冲算法。

```
        session.createCriteria(User.class)
                .add( Restrictions.naturalId()
                        .set("name", "gavin")
                        .set("org", "hb")
```

```
                ).setCacheable(true)
                .uniqueResult();
```

## 参考


[PoEAA] *Patterns of Enterprise Application Architecture（企业应用架构模式）*. 0-321-12742-0. Martin Fowler. Copyright © 2003 Pearson Education, Inc.. Addison-Wesley Publishing Company.

[JPwH] *Java Persistence with Hibernate（Hibernate 的Java 持久化）*. Second Edition of Hibernate in Action（Hibernate in Action 第二版）. 1-932394-88-5. http://www.manning.com/bauer2 . Christian Bauer and Gavin King. Copyright © 2007 Manning Publications Co.. Manning Publications Co..