



####

Version: 3.2.0 CR1

目录

前言	iv
1. 翻译说明	iv
2. 版权声明	v
前言	vi
1. 创建一个注解项目	1
1.1. 系统需求	1
1.2. 系统配置	1
2. 实体Bean	3
2.1. 简介	3
2.2. 用EJB3注解进行映射	3
2.2.1. 声明实体bean	3
2.2.1.1. 定义表(Table)	3
2.2.1.2. 乐观锁定版本控制	4
2.2.2. 映射简单属性	4
2.2.2.1. 声明基本的属性映射	4
2.2.2.2. 声明列属性	6
2.2.2.3. 嵌入式对象(又名组件)	7
2.2.2.4. 无注解之属性的默认值	8
2.2.3. 映射主键属性	8
2.2.4. 映射继承关系	11
2.2.4.1. 每个类一张表	12
2.2.4.2. 每个类层次结构一张表	12
2.2.4.3. 连接的子类	13
2.2.4.4. 从父类继承的属性	13
2.2.5. 映射实体Bean的关联关系	14
2.2.5.1. 一对一(One-to-one)	14
2.2.5.2. 多对一(Many-to-one)	16
2.2.5.3. 集合类型	17
2.2.5.4. 用cascading实现传播性持久化(Transitive persistence)	23
2.2.5.5. 关联关系获取	23
2.2.6. 映射复合主键与外键	23
2.2.7. 映射二级表(secondary tables)	25
2.3. 映射查询	26
2.3.1. 映射EJBQL/HQL查询	26
2.3.2. 映射本地化查询	27
2.4. Hibernate独有的注解扩展	30
2.4.1. 实体	30
2.4.2. 标识符	32
2.4.3. 属性	32
2.4.3.1. 访问类型	32
2.4.3.2. 公式	33
2.4.3.3. 类型	34
2.4.3.4. 索引	35
2.4.3.5. @Parent	35

2.4.3.6. 生成的属性	35
2.4.4. 继承	35
2.4.5. 关于单个关联关系的注解	36
2.4.5.1. 延迟选项和获取模式	36
2.4.6. 关于集合类型的注解	37
2.4.6.1. 参数注解	37
2.4.6.2. 更多的集合类型	37
2.4.7. 缓存	40
2.4.8. 过滤器	41
2.4.9. 查询	41
3. 通过XML覆写元数据	42
3.1. 原则	42
3.1.1. 全局级别的元数据	42
3.1.2. 实体级别的元数据	42
3.1.3. 属性级别的元数据	45
3.1.4. 关联级别的元数据	45
4. Hibernate验证器	47
4.1. 约束	47
4.1.1. 什么是约束?	47
4.1.2. 内建约束	47
4.1.3. 错误信息	48
4.1.4. 编写你自己的约束	48
4.1.5. 注解你的领域模型	50
4.2. 使用验证器框架	51
4.2.1. 数据库schema层次验证	51
4.2.2. Hibernate基于事件的验证	52
4.2.3. 程序级验证	52
4.2.4. 验证信息	53
5. Hibernate与Lucene集成	54
5.1. 使用Lucene为实体建立索引	54
5.1.1. 注解领域模型	54
5.1.2. 启用自动索引	54
A. 术语表	56

前言

WARNING! This is a translated version of the English Hibernate reference documentation. The translated version might not be up to date! However, the differences should only be very minor. Consult the English reference documentation if you are missing information or encounter a translation error. If you like to contribute to a particular translation, contact us on the Hibernate developer mailing list.

Translator(s): RedSaga Translate Team 满江红翻译团队 <caoxg@yahoo.com>

1. 翻译说明

本文档的翻译是在网络上协作进行的，也会不断根据Hibernate的升级进行更新。提供此文档的目的是为了减缓学习Hibernate的坡度，而非代替原文档。我们建议所有有能力的读者都直接阅读英文原文。若您对翻译有异议，或发现翻译错误，敬请不吝赐教，报告到如下地址：
<http://wiki.redsaga.com/confluence/display/HART/Home>

表 1. Hibernate Annotation v3翻译团队

序号	标题	中文标题	翻译	1审	2审
--	Contents	目录	Liu Chang		
#1	Setting up an annotations projec	创建一个注解项目	melthaw	Zheng Shuai	superq
#2	Entity Beans-Introduction	实体Bean-简介	melthaw	Zheng Shuai	superq
#3	Entity Beans-Mapping with EJB3 Annotations	实体Bean-用EJB3注解进行映射	melthaw	Zheng Shuai	superq, Liu Chang, Sean Chan
#4	Entity Beans-Mapping Queries	实体Bean-映射查询	melthaw	Zheng Shuai	superq, Liu Chang, Sean Chan
#5	Entity Beans-Hibernate Annotation Extensions	实体Bean-Hibernate独有的注解扩展	Sean Chan	morning	melthaw
#6	Overriding metadata through XML	通过XML覆写元数据	icess	melthaw	Sean Chan

序号	标题	中文标题	翻译	1审	2审
#7	Hibernate Validator	Hibernate验证器	DigitalSonic	morning	melthaw
#8	Hibernate Lucene Integration	Hibernate 与 Lucene集成	mochow	morning	melthaw
#9	Appendix:Glossary	附录:术语表	mochow	Liu Chang	曹晓钢

关于我们

满江红. 开源, <http://www.redsaga.com>

从成立之初就致力于Java开放源代码在中国的传播与发展, 与国内多个Java团体及出版社有深入交流。坚持少说多做的原则, 目前有两个团队, “OpenDoc团队”与“翻译团队”, 本翻译文档即为翻译团队作品。OpenDoc团队已经推出包括Hibernate、iBatis、Spring、WebWork的多份开放文档, 并于2005年5月在Hibernate开放文档基础上扩充成书, 出版了原创书籍:《深入浅出Hibernate》, 本书400余页, 适合各个层次的Hibernate用户。
(http://www.redsaga.com/hibernate_book.html)敬请支持。

致谢

在我们翻译Hibernate Annotation参考文档的同时, 还有一位热心的朋友也在进行着同样的工作, 这位朋友就是 icess(冰雨), 由 icess 翻译的中文版的地址:
<http://icess.my.china.com/hibernate/a/ref/index.htm>

2. 版权声明

Hibernate英文文档属于Hibernate发行包的一部分, 遵循LGPL协议。本翻译版本同样遵循LGPL协议。参与翻译的译者一致同意放弃除署名权外对本翻译版本的其它权利要求。

您可以自由链接、下载、传播此文档, 或者放置在您的网站上, 甚至作为产品的一部分发行。但前提是必须保证全文完整转载, 包括完整的版权信息和作译者声明, 并不能违反LGPL协议。这里“完整”的含义是, 不能进行任何删除/增添/注解。若有删除/增添/注解, 必须逐段明确声明那些部分并非本文档的一部分。

前言

正如其他的ORM工具, Hibernate同样需要元数据来控制在不同数据表达形式之间的转化. 在Hibernate 2.x里, 多数情况下表示映射关系的元数据保存在XML文本文件中. 还有一种方式就是Xdoclet, 它可以在编译时利用Javadoc中的源码注释信息来进行预处理. 现在新的JDK标准 (JDK1.5以上) 也支持类似的注解功能, 但相比之下很多工具对此提供了更强大更好用的支持. 以IntelliJ IDEA和Eclipse为例, 这些IDE工具为JDK 5.0注解功能提供了自动完成和语法高亮功能. 注解被直接编译到字节码里, 并在运行时 (对于Hibernate来讲就是启动的时候) 通过反射读取这些注解, 因此外部XML文件就不再需要了.

EJB3规范最终认可了透明化ORM的成功范例以及市场对于这种技术的兴趣. EJB3规范标准化了ORM的基础API而且在任何ORM持久化机制中使用元数据. Hibernate EntityManager实现了EJB3持久化规范中定义的编程接口和生命周期规则. 在Hibernate Core的基础上再结合 Hibernate Annotations就实现了一套完整 (并且独立) 的EJB3持久化解决方案. 你可以结合三者来使用, 也可以抛开EJB3编程接口和生命周期规则而独立使用注解, 甚至只单独使用Hibernate Core. 这些都取决于项目的商业和技术上的实际需求. Hibernate允许你直接使用native APIs, 如果有需要, 甚至可以直接操作JDBC和SQL.

注意本文档基于Hibernate Annotations的预览版 (遵从EJB 3.0/JSR-220最终草案). 这个版本和新规范中定义的最终概念已经非常接近了. 我们的目标是提供一套完整的ORM注解, 包括EJB3的标准注解以及Hibernate3的扩展 (后者是EJB3规范中没有涉及到的). 最终通过注解你可以完成任何可能的映射. 详情参考???

EJB3最终草案修改了部分注解, <http://www.hibernate.org/371.html>提供了从上一个版本到最新版本的迁移指南.

第 1 章 创建一个注解项目

1.1. 系统需求

- 首先从Hibernate官方网站下载并解压Hibernate Annotations的发布包。
- 这个版本(预览版)要求使用Hibernate 3.2.0.CR2或更高版本。请不要和老版本的Hibernate 3.x混合起来使用。
- 这个版本在Hibernate core 3.2.0.CR2的基础上工作良好。
- 首先确定你已经安装了JDK 5.0。当然就算使用低版本的JDK，Xdoclet也可以提供（基于注解的）元数据所带来的部分功能。不过请注意本文档只描述跟JDK5.0注解有关的内容，关于Xdoclet请参考相关文档。

1.2. 系统配置

首先就是设置classpath(当然是在IDE中创建了一个新项目之后)。

- 将Hibernate3核心文件以及其依赖的第三方库文件(请参考lib/README.txt文件)加入到你的classpath里面。
- 将hibernate-annotations.jar 和lib/ejb3-persistence.jar加入到你的classpath里面。
- 如果要使用 第 5 章 Hibernate与Lucene集成，还需要将lucene的jar文件加入你的classpath。

我们推荐在一个包装器(wrapper)类HibernateUtil 的静态初始化代码块中启动Hibernate。或许你在Hibernate文档的其他很多地方看到过这个类，但是要在你的项目中使用注解，还需要对这个辅助(helper)类进行扩展。扩展如下：

```
package hello;

import org.hibernate.*;
import org.hibernate.cfg.*;
import test.*;
import test.animals.Dog;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {

            sessionFactory = new AnnotationConfiguration().buildSessionFactory();
        } catch (Throwable ex) {
            // Log exception!
            throw new ExceptionInInitializerError(ex);
        }
    }
}
```

```

public static Session getSession()
    throws HibernateException {
    return sessionFactory.openSession();
}
}

```

这里比较有意思的是使用到了AnnotationConfiguration类。在XML配置文件(通常是hibernate.cfg.xml)中则定义了包和经过注解的类。下面的xml和前面的声明等价:

```

<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

    <hibernate-configuration>
        <session-factory>
            <mapping package="test.animals"/>
            <mapping class="test.Flight"/>
            <mapping class="test.Sky"/>
            <mapping class="test.Person"/>
            <mapping class="test.animals.Dog"/>
            <mapping resource="test/animals/orm.xml"/>
        </session-factory>
    </hibernate-configuration>

```

注意现在你可以混合使用hbm.xml和注解。资源元素(resource element)可以是hbm文件也可以是EJB3 XML发布描述符,此差别对于配置过程是透明的。

除了上面的方式,你还可以通过编程的方式定义包括注解的类和包

```

sessionFactory = new AnnotationConfiguration()
    .addPackage("test.animals") //the fully qualified package name
    .addAnnotatedClass(Flight.class)
    .addAnnotatedClass(Sky.class)
    .addAnnotatedClass(Person.class)
    .addAnnotatedClass(Dog.class)
    .buildSessionFactory();

```

你也可以使用Hibernate Entity Manager来完成以上功能。Hibernate Entity Manager有自己的一套配置机制,详情请参考相关文档。

除了启动方式和配置文件有所改变之外,结合注解来使用Hibernate API和以前没有什么区别,在其他方面你还是可以继续保持以前的习惯和喜好(hibernate.properties, hibernate.cfg.xml, programmatic APIs等等)。甚至对于同一个SessionFactory,你都可以混合带注解的持久类以及传统的bm.cfg.xml声明方式。然而你不能多次声明同一个类(要么通过注解要么通过hbm.xml配置文件),而且在一个映射实体的类继承层次中,这两个配置策略不能同时使用。

为了简化从hbm文件到注解的迁移过程,配置机制将自动检测在注解和hbm文件中重复的映射。默认情况下hbm文件中的声明比类中的注解元数据具有更高的优先级。这种优先级的设定是以类为单位的。你也可以通过hibernate.mapping.precedence修改这种优先级。默认的值是hbm, class, 如果改为class, hbm, 当发生冲突的时候,类中的注解将比hbm文件具有更高的优先级。

第 2 章 实体Bean

2.1. 简介

本章内容覆盖了EJB3.0实体bean的注解规范以及Hibernate特有的扩展。

2.2. 用EJB3注解进行映射

现在EJB3实体Bean是纯粹的POJO。实际上这表达了和Hibernate持久化实体对象同样的概念。它们的映射都通过JDK5.0注解来定义(EJB3规范中的XML描述语法至今还没有最终定下来)。注解分为两个部分, 分别是逻辑映射注解和物理映射注解, 通过逻辑映射注解可以描述对象模型, 类之间的关系等等, 而物理映射注解则描述了物理的schema, 表, 列, 索引等等。下面我们在代码中将混合使用这两种类型的注解。

EJB3注解的API定义在`javax.persistence.*`包里面。大部分和JDK5兼容的IDE(象Eclipse, IntelliJ IDEA和Netbeans等等)都提供了注解接口和属性的自动完成功能。(这些不需要IDE提供特别的EJB3支持模块, 因为EJB3注解是标准的JDK5注解)

请阅读JBoss EJB 3.0指南或者直接阅读Hibernate Annotations测试代码以获取更多的可运行实例。Hibernate Annotations提供的大部分单元测试代码都演示了实际的例子, 是一个获取灵感的好地方。

2.2.1. 声明实体bean

每一个持久化POJO类都是一个实体bean, 这可以通过在类的定义中使用@Entity注解来进行声明:

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

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

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

通过@Entity注解将一个类声明为一个实体bean(即一个持久化POJO类), @Id注解则声明了该实体bean的标识属性。其他的映射定义是隐式的。这种以隐式映射为主体, 以显式映射为例外的配置方式在新的EJ3规范中处于非常重要的位置, 和以前的版本相比有了质的飞跃。在上面这段代码中: Flight类映射到Flight表, 并使用id列作为主键列。

在对一个类进行注解时, 你可以选择对它的属性或者方法进行注解, 根据你的选择, Hibernate的访问类型分别为 field或property。EJ3规范要求需要在需要访问的元素上进行注解声明, 例如, 如果访问类型为property就要在getter方法上进行注解声明, 如果访问类型为 field就要在字段上进行注解声明。应该尽量避免混合使用这两种访问类型。Hibernate根据@Id 或 @EmbeddedId的位置来判断访问类型。

2.2.1.1. 定义表(Table)

@Table是类一级的注解，通过@Table注解可以为实体bean映射指定表(table)，目录(catalog)和schema的名字。如果没有定义@Table，那么系统自动使用默认值：实体的短类名(不附带包名)。

```
@Entity
@Table(name="tbl_sky")
public class Sky implements Serializable {
    ...
}
```

@Table 元素包括了一个schema 和一个 catalog 属性，如果需要可以指定相应的值。结合使用@UniqueConstraint注解可以定义表的唯一约束(unique constraint)（对于绑定到单列的唯一约束，请参考@Column注解）

```
@Table(name="tbl_sky",
        uniqueConstraints = {@UniqueConstraint(columnNames={"month", "day"})})
```

上面这个例子中，在month和day这两个字段上定义唯一约束。注意columnNames数组中的值指的是逻辑列名。

Hibernate在NamingStrategy的实现中定义了逻辑列名。默认EJB3命名策略将物理字段名当作逻辑字段名来使用。注意该字段名和它对应的属性名可能不同(如果字段名是显式指定的话)。除非你重写了NamingStrategy，否则不用担心这些区别。

2.2.1.2. 乐观锁定版本控制

你可以在实体bean中使用@Version注解，通过这种方式可添加对乐观锁定的支持：

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

上面这个例子中，version属性将映射到OPTLOCK列，entity manager使用该字段来检测更新冲突(防止更新丢失，请参考last-commit-wins策略)。

根据EJB3规范，version列可以是numeric类型(推荐方式)也可以是timestamp类型。Hibernate支持任何自定义类型，只要该类型实现了UserVersionType。

2.2.2. 映射简单属性

2.2.2.1. 声明基本的属性映射

Every non static non transient property (field or method) of an entity bean is considered persistent, unless you annotate it as @Transient. Not having an annotation for your property is equivalent to the appropriate @Basic annotation. The @Basic annotation allows you to declare the fetching strategy for a property:

实体bean中所有的非static非transient的属性都可以被持久化, 除非你将其注解为@Transient. 所有没有定义注解的属性等价于在其上面添加了@Basic注解. 通过 @Basic注解可以声明属性的获取策略(fetch strategy):

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

private String firstname; //persistent property

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

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

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

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

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

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

上面这个例子中, counter 是一个transient的字段, lengthInMeter的getter方法被注解为@Transient, entity manager将忽略这些字段和属性. 而name, length, firstname 这几个属性则是被定义为可持久化和可获取的. 对于简单属性来说, 默认的获取方式是即时获取(early fetch). 当一个实体Bean的实例被创建时, Hibernate会将这些属性的值从数据库中提取出来, 保存到Bean的属性里. 与即时获取相对应的是延迟获取(lazy fetch). 如果一个属性的获取方式是延迟获取 (比如上面例子中的detailedComment属性), Hibernate在创建一个实体Bean的实例时, 不会即时将这个属性的值从数据库中读出. 只有在该实体Bean的这个属性第一次被调用时, Hibernate才会去获取对应的值. 通常你不需要对简单属性设置延迟获取(lazy simple property), 千万不要和延迟关联获取(lazy association fetch)混淆了 (译注: 这里指不要把lazy simple property和lazy association fetch混淆了).

注意

为了启用属性级的延迟获取, 你的类必须经过特殊处理(instrumented): 字节码将被织入原始类中来实现延迟获取功能, 详情参考Hibernate参考文档. 如果不对类文件进行字节码特殊处理, 那么属性级的延迟获取将被忽略.

推荐的替代方案是使用EJB-QL或者Criteria查询的投影(projection)功能.

Hibernate和EJB3都支持所有基本类型的属性映射. 这些基本类型包括所有的Java基本类型, 及其各自的wrapper类和serializable类. Hibernate Annotations还支持将内置的枚举类型映射到一个顺序列(保存了相应的序列值) 或一个字符串类型的列(保存相应的字符串). 默认是保存枚举的序列值, 但是您可以通过@Enumerated注解来进行调整(见上面例子中的note属性).

在核心的Java API中并没有定义时间精度(temporal precision). 因此处理时间类型数据时, 你还需要定义将其存储在数据库中所预期的精度. 在数据库中, 表示时间类型的数据有DATE, TIME, 和 TIMESTAMP三种精度(即单纯的日期, 时间, 或者两者兼备). 可使用@Temporal注解来调整精度.

@Lob注解表示属性将被持久化为Blob或者Clob类型, 具体取决于属性的类型, java.sql.Clob, Character[],

`char[]` 和 `java.lang.String` 这些类型的属性都被持久化为 `Clob` 类型，而 `java.sql.Blob`, `Byte[]`, `byte[]` 和 `serializable` 类型则被持久化为 `Blob` 类型。

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

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

如果某个属性实现了 `java.io.Serializable` 同时也不是基本类型，并且没有在该属性上使用 `@Lob` 注解，那么 Hibernate 将使用自带的 `serializable` 类型。

2.2.2.2. 声明列属性


使用 `@Column` 注解可将属性映射到列。使用该注解来覆盖默认值(关于默认值请参考EJB3规范)。在属性级使用该注解的方式如下：

- 不进行注解
- 和 `@Basic` 一起使用
- 和 `@Version` 一起使用
- 和 `@Lob` 一起使用
- 和 `@Temporal` 一起使用
- 和 `@org.hibernate.annotations.CollectionOfElements` 一起使用（只针对Hibernate）

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

在上面这个例子中，`name` 属性映射到 `flight_name` 列。该字段不允许为空，长度为50，并且是不可更新的（也就是属性值是不变的）。

上面这些注解可以被应用到正规属性上例如 `@Id` 或 `@Version` 属性。

```
@Column( 
    name="columnName";                                (1)
    boolean unique() default false;                   (2)
    boolean nullable() default true;                   (3)
    boolean insertable() default true;                 (4)
    boolean updatable() default true;                  (5)
    String columnDefinition() default "";              (6)
    String table() default "";                         (7)
```

```
int length() default 255; (8)
int precision() default 0; // decimal precision (9)
int scale() default 0; // decimal scale
```

- (1) name 可选, 列名(默认值是属性名)
- (2) unique 可选, 是否在该列上设置唯一约束(默认值false)
- (3) nullable 可选, 是否设置该列的值可以为空(默认值false)
- (4) insertable 可选, 该列是否作为生成的insert语句中的一个列(默认值true)
- (5) updatable 可选, 该列是否作为生成的update语句中的一个列(默认值true)
- (6) columnDefinition 可选: 为这个特定列覆盖SQL DDL片段 (这可能导致无法在不同数据库间移植)
- (7) table 可选, 定义对应的表(默认为主表)
- (8) length 可选, 列长度(默认值255)
- (8) precision 可选, 列十进制精度(decimal precision)(默认值0)
- (10) scale 可选, 如果列十进制数值范围(decimal scale)可用, 在此设置(默认值0)

2.2.2.3. 嵌入式对象(又名组件)

在实体中可以定义一个嵌入式组件(embedded component), 甚至覆盖该实体中原有的列映射. 组件类必须在类一级定义@Embeddable注解. 在特定的实体的关联属性上使用@Embedded和 @AttributeOverride注解可以覆盖该属性对应的嵌入式对象的列映射:

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
    Address homeAddress;

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

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

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

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

```

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

```

嵌入式对象继承其所属实体中定义的访问类型（注意：这可以通过使用Hibernate提供的@AccessType注解来覆盖原有值）（请参考 Hibernate Annotation Extensions）。

在上面的例子中，实体bean Person 有两个组件属性，分别是homeAddress和bornIn。我们可以看到homeAddress 属性并没有注解。但是Hibernate自动检测其对应的Address类中的@Embeddable注解，并将其看作一个持久化组件。对于Country中已映射的属性，则使用@Embedded和@AttributeOverride 注解来覆盖原来映射的列名。正如你所看到的，Address对象中还内嵌了Country对象，这里和homeAddress一样使用了Hibernate和EJB3自动检测机制。目前EJB3规范还不支持覆盖多层嵌套（即嵌入式对象中还包括其他嵌入式对象）的列映射。不过Hibernate通过在表达式中使用“.”符号表达式提供了对此特征的支持。

```

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

```

Hibernate 注解支持很多EJB3 规范中没有明确定义的特性。例如，可以在嵌入式对象上添加@MappedSuperclass注解，这样可以将其父类的属性持久（详情请查阅@MappedSuperclass）。

Hibernate现在支持在嵌入式对象中使用关联注解（如@*ToOne和@*ToMany）。而EJB3规范尚不支持这样的用法。你可以使用 @AssociationOverride注解来覆写关联列。

在同一个实体中使用两个同类型的嵌入对象，其默认列名是无效的：至少要对其中一个进行明确声明。Hibernate在这方面走在了EJB3规范的前面，Hibernate提供了NamingStrategy，在使用Hibernate时，通过NamingStrategy 你可以对默认的机制进行扩展。DefaultComponentSafeNamingStrategy 在默认的EJB3NamingStrategy上进行了小小的提升，允许在同一实体中使用两个同类型的嵌入对象而无须额外的声明。

2.2.2.4. 无注解之属性的默认值

如果某属性没有注解，该属性将遵守下面的规则：

- 如果属性为单一类型，则映射为@Basic
- 否则，如果属性对应的类型定义了@Embeddable注解，则映射为@Embedded
- 否则，如果属性对应的类型实现了Serializable，则属性被映射为@Basic并在一个列中保存该对象的serialized版本
- 否则，如果该属性的类型为java.sql.Clob 或 java.sql.Blob，则作为@Lob并映射到适当的LobType。

2.2. . 映射主键属性

使用@Id注解可以将实体bean中的某个属性定义为标识符(identifier)。该属性的值可以通过应用自身进行设置，也可以通过Hibernate生成(推荐)。使用 @GeneratedValue注解可以定义该标识符的生成策略：

- AUTO - 可以是identity column类型, 或者sequence类型或者table类型, 取决于不同的底层数据库。
- TABLE - 使用表保存id值
- IDENTITY - identity column
- SEQUENCE - sequence

和EJB3规范相比, Hibernate提供了更多的id生成器. 详情请查阅 [Hibernate Annotation Extensions](#) .

下面的例子展示了使用SEQ_STORE配置的sequence生成器

```
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_STORE")
public Integer getId() { ... }
```

下面这个例子使用的是identity生成器

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
public Long getId() { ... }
```

AUTO生成器适用于可移植的应用(在多个DB间切换)。多个@Id可以共享同一个identifier生成器, 只要把generator属性设成相同的值就可以了。通过@SequenceGenerator 和 @TableGenerator, 你可以配置不同的identifier生成器。每一个identifier生成器都有自己的适用范围, 可以是应用级(application level)和类级(class level)。类一级的生成器在外部是不可见的, 而且类一级的生成器可以覆盖应用级的生成器。应用级的生成器则定义在包一级(package level) (如package-info.java) :

```
@javax.persistence.TableGenerator(
    name="EMP_GEN",
    table="GENERATOR_TABLE",
    pkColumnName = "key",
    valueColumnName = "hi",
    pkColumnValue="EMP",
    allocationSize=20
)
@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence"
)
package org.hibernate.test.metadata;
```

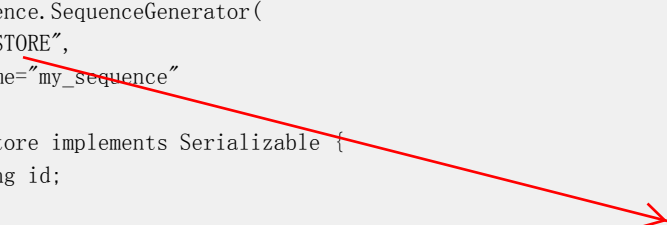
如果在org.hibernate.test.metadata包下面的 package-info.java文件用于初始化EJB配置, 那么该文件中定义的 EMP_GEN 和SEQ_GEN都是应用级的生成器。EMP_GEN定义了一个使用hilo算法 (max_lo为20)的id生成器(该生成器将id的信息存在数据库的某个表中)。id的hi值保存在GENERATOR_TABLE中。在该表中pkColumnName "key"等价于 pkColumnValue "EMP", 而valueColumnName "hi"中存储的是下一个要使用的最大值。

SEQ_GEN 则定义了一个sequence 生成器, 其对应的sequence名为 my_sequence. 注意目前Hibernate Annotations还不支持sequence 生成器中的 initialValue和 allocationSize参数.


下面这个例子展示了定义在类范围(class scope)的sequence生成器:

```
@Entity
@javax.persistence.SequenceGenerator(
    name="SEQ_STORE",
    sequenceName="my_sequence"
)
public class Store implements Serializable {
    private Long id;

    @Id @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_STORE")
    public Long getId() { return id; }
}
```



在这个例子中, Store类使用名为my_sequence的sequence, 并且SEQ_STORE 生成器对于其他类是不可见的. 注意在org.hibernate.test.metadata.id包下的测试代码有更多演示Hibernate Annotations用法的例子..

下面是定义组合主键的几种语法: 

- 将组件类注解为@Embeddable, 并将组件的属性注解为@Id
- 将组件的属性注解为@EmbeddedId
- 将类注解为@IdClass, 并将该实体中所有属于主键的属性都注解为@Id

对于EJB2的开发人员来说 @IdClass是很常见的, 但是对于Hibernate的用户来说就是一个崭新的用法. 组合主键类对应了一个实体类中的多个字段或属性, 而且主键类中用于定义主键的字段或属性和 实体类中对应的字段或属性在类型上必须一致. 下面我们看一个例子:

```
@Entity
@IdClass(FootballerPk.class)
public class Footballer {
    //part of the id key
    @Id public String getFirstname() {
        return firstname;
    }

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

    //part of the id key
    @Id public String getLastName() {
        return lastname;
    }

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

    public String getClub() {
```



```

        return club;
    }

    public void setClub(String club) {
        this.club = club;
    }

    //appropriate equals() and hashCode() implementation
}

@Embeddable
public class FootballerPk implements Serializable {
    //same name and type as in Footballer
    public String getFirstname() {
        return firstname;
    }

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

    //same name and type as in Footballer
    public String getLastname() {
        return lastname;
    }

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

    //appropriate equals() and hashCode() implementation
}

```

如上，@IdClass指向对应的主键类。

Hibernate支持在组合标识符中定义关联(就像使用普通的注解一样)，而EJB3规范并不支持此类用法。

```

@Entity
@AssociationOverride( name="id.channel", joinColumns = @JoinColumn(name="chan_id") )
public class TvMagazin {
    @EmbeddedId public TvMagazinPk id;
    @Temporal(TemporalType.TIME) Date time;
}

@Embeddable
public class TvMagazinPk implements Serializable {
    @ManyToOne
    public Channel channel;
    public String name;
    @ManyToOne
    public Presenter presenter;
}

```

2.2.4. 映射继承关系

EJB3支持三种类型的继承映射：

- 每个类一张表(Table per class)策略: 在Hibernate中对应<union-class>元素:
- 每个类层次结构一张表(Single table per class hierarchy)策略: 在Hibernate中对应<subclass>元素
- 连接的子类(Joined subclasses)策略: 在Hibernate中对应 <joined-subclass>元素

你可以用 `@Inheritance` 注解来定义所选择的策略. 这个注解需要在每个类层次结构(class hierarchy)最顶端的实体类上使用.

注意

目前还不支持在接口上进行注解.

2.2.4.1. 每个类一张表

这种策略有很多缺点(例如:多态查询和关联), EJB3规范, Hibernate参考手册, Hibernate in Action, 以及其他许多地方都对此进行了描述和解释. Hibernate使用SQL UNION查询来实现这种策略. 通常使用场合是在一个继承层次结构的顶端:

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

这种策略支持双向的一对多关联. 这里不支持IDENTITY生成器策略, 因为id必须在多个表间共享. 当然, 一旦使用这种策略就意味着你不能使用 AUTO 生成器和IDENTITY生成器.

2.2.4.2. 每个类层次结构一张表

整个继承层次结构中的父类和子类的所有属性都映射到同一个表中, 他们的实例通过一个辨别符(discriminator)列来区分.:

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

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

在上面这个例子中, Plane是父类, 在这个类里面将继承策略定义为 `InheritanceType.SINGLE_TABLE`, 并通过 `@DiscriminatorColumn` 注解定义了辨别符列(还可以定义辨别符的类型). 最后, 对于继承层次结构中的每个类, `@DiscriminatorValue` 注解指定了用来辨别该类的值. 辨别符列的名字默认为 `DTYPE`, 其默认值为实体名(在 `@Entity.name` 中定义), 其类型为 `DiscriminatorType.STRING`. A320是子类, 如果不想使用默认的辨别

符, 只需要指定相应的值即可. 其他的如继承策略, 辨别标志字段的类型都是自动设定的.

`@Inheritance` 和 `@DiscriminatorColumn` 注解只能用于实体层次结构的顶端.

2.2.4.3. 连接的子类

当每个子类映射到一个表时, `@PrimaryKeyJoinColumn` 和 `@PrimaryKeyJoinColumns` 注解定义了每个子类表关联到父类表的主键:

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

@Entity
public class Ferry extends Boat { ... }

@Entity
@PrimaryKeyJoinColumn(name="BOAT_ID")
public class AmericaCupClass extends Boat { ... }
```

以上所有实体都使用了JOINED策略, Ferry表和Boat表使用同名的主键. 而AmericaCupClass表和Boat表使用了条件 `Boat.id = AmericaCupClass.BOAT_ID`进行关联.

2.2.4.4. 从父类继承的属性

有时候通过一个(技术上或业务上)父类共享一些公共属性是很有用的, 同时还不用将该父类作为映射的实体(也就是该实体没有对应的表). 这个时候你需要使用`@MappedSuperclass`注解来进行映射.

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

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

在数据库中, 上面这个例子中的继承的层次结构最终以Order表的形式出现, 该表拥有id, lastUpdate 和 lastUpdater三个列. 父类中的属性映射将复制到其子类实体. 注意这种情况下的父类不再处在继承层次结构的顶端.

注意

注意, 没有注解为`@MappedSuperclass`的父类中的属性将被忽略.

注意

除非显式使用Hibernate annotation中的`@AccessType`注解, 否则将从继承层次结构的根实体中继

承访问类型(包括字段或方法)

注意

这对于`@Embeddable`对象的父类中的属性持久化同样有效。只需要使用`@MappedSuperclass`注解即可（虽然这种方式不会纳入EJB3标准）

注意

可以将处在映射继承层次结构的中间位置的类注解为`@MappedSuperclass`。

注意

在继承层次结构中任何没有被注解为`@MappedSuperclass` 或`@Entity`的类都将被忽略。

你可以通过 `@AttributeOverride`注解覆盖实体父类中的定义的列。这个注解只能在继承层次结构的顶端使用。

```
@MappedSuperclass
public class FlyingObject implements Serializable {

    public int getAltitude() {
        return altitude;
    }

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

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

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

在上面这个例子中, `altitude`属性的值最终将持久化到`Plane` 表的`fld_altitude`列. 而名为`propulsion`的关联则保存在`fld_propulsion_fk`外间列.

你可以为`@Entity`和`@MappedSuperclass`注解的类 以及那些对象为`@Embeddable`的属性定义 `@AttributeOverride`和`@AssociationOverride`.

2.2.5. 映射实体Bean的关联关系

2.2.5.1. 一对一 (One-to-one)

使用`@OneToOne`注解可以建立实体bean之间的一对一的关联。一对一关联有三种情况：一是关联的实体

都共享同样的主键，二是其中一个实体通过外键关联到另一个实体的主键（注意要模拟一对一关联必须在外键列上添加唯一约束）。三是通过关联表来保存两个实体之间的连接关系（注意要模拟一对一关联必须在每一个外键上添加唯一约束）。

首先, 我们通过共享主键来进行一对一关联映射:



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

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

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

上面的例子通过使用注解@PrimaryKeyJoinColumn定义了一对一关联。

下面这个例子使用外键列进行实体的关联。



```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="passport_fk")
    public Passport getPassport() {
        ...
    }
}

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
        ...
    }
}
```

上面这个例子中, Customer 通过Customer 表中名为的passport_fk 外键列和 Passport关联. @JoinColumn注解定义了联接列(join column). 该注解和@Column注解有点类似, 但是多了一个名为referencedColumnName的参数. 该参数定义了所关联目标实体中的联接列. 注意, 当referencedColumnName关联到非主键列的时候, 关联的目标类必须实现Serializable, 还要注意的是所映射的属性对应单个列(否则映射无效).

一对一关联可能是双向的. 在双向关联中, 有且仅有一端是作为主体(owner)端存在的: 主体端负责维护联接列(即更新). 对于不需要维护这种关系的从表则通过mappedBy属性进行声明. mappedBy的值指向

主体的关联属性 在上面这个例子中, mappedBy的值为 passport. 最后, 不必也不能再在被关联端(owned side)定义联接列了, 因为已经在主体端进行了声明.

如果在主体没有声明@JoinColumn, 系统自动进行处理: 在主表(owner table)中将创建联接列, 列名为: 主体的关联属性名+下划线+被关联端的主键列名. 在上面这个例子中是passport_id, 因为Customer中关联属性名为passport, Passport的主键是id.

The third possibility (using an association table) is very exotic.

第三种方式也许是最另类的(通过关联表).

```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinTable(name = "CustomerPassports",
        joinColumns = @JoinColumn(name="customer_fk"),
        inverseJoinColumns = @JoinColumn(name="passport_fk")
    )
    public Passport getPassport() {
        ...
    }
}

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
        ...
    }
}
```

Customer通过名为 CustomerPassports的关联表和 Passport关联; 该关联表拥有名为passport_fk的外键列, 该外键指向Passport表, 该信息定义为inverseJoinColumn的属性值, 而customer_fk外键列指向Customer表, 该信息定义为 joinColumns的属性值.

这种关联可能是双向的. 在双向关联中, 有且仅有一端是作为主体端存在的: 主体端负责维护联接列(即更新). 对于不需要维护这种关系的从表则通过mappedBy属性进行声明. mappedBy的值指向主体的关联属性. 在上面这个例子中, mappedBy的值为 passport. 最后, 不必也不能再在被关联端(owned side)定义联接列了, 因为已经在主体端进行了声明.

你必须明确定义关联表名和关联列名.

2.2.5.2. 多对一 (Many-to-one)

在实体属性一级使用@ManyToOne注解来定义多对一关联:

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

其中@JoinColumn是可选的, 关联字段默认值和一对一 (one to one) 关联的情况相似, 列名为: 主体的关联属性名+下划线+被关联端的主键列名. 在这个例子中是company_id, 因为关联的属性是company, Company的主键是id.

@ManyToOne注解有一个名为targetEntity的参数, 该参数定义了目标实体名. 通常不需要定义该参数, 因为在大部分情况下默认值(表示关联关系的属性类型)就可以很好的满足要求了. 不过下面这种情况下这个参数就显得有意义了: 使用接口作为返回值而不是常见的实体.

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

public interface Company {
    ...
}
```

对于多对一也可以通过关联表的方式来映射。通过@JoinTable注解可定义关联表, 该关联表包含了指向实体表的外键(通过@JoinTable.joinColumns) 以及指向目标实体表的外键(通过@JoinTable.inverseJoinColumns)。

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

2.2.5.3. 集合类型

2.2.5.3.1. 概况

你可以对 Collection ,List (指有序列表, 而不是索引列表), Map和Set这几种类型进行映射. EJB3规范定义了怎么样使用@javax.persistence.OrderBy 注解来对有序列表进行映射: 该注解接受的参数格式: 用逗号隔开的(目标实体)属性名及排序指令, 如firstname asc, age desc, 如果该参数为空, 则默认以id对该集合进行排序. 如果某个集合在数据库中对应一个关联表(association table)的话, 你不能在这个集合属性上面使用@OrderBy注解. 对于这种情况的处理方法, 请参考Hibernate Annotation Extensions. EJB3 允许你利用目标实体的一个属性作为Map的key, 这个属性可以用@MapKey(name="myProperty")来声明. 如果使用@MapKey注解的时候不提供属性名, 系统默认使用目标实体的主键. map的key使用和属性相同的列: 不需要为map key定义专用的列, 因为map key实际上就表达了一个目标属性. 注意一旦加载, key

不再和属性保持同步，也就是说，如果你改变了该属性的值，在你的Java模型中的key不会自动更新（请参考Hibernate Annotation Extensions）。很多人被<map>和@MapKey弄糊涂了。其他它们有两点区别。@MapKey目前还有一些限制，详情请查看论坛或者 我们的JIRA缺陷系统。注意一旦加载，key不再和属性保持同步，也就是说，如果你改变了该属性的值，在你的Java模型中的key不会自动更新。（Hibernate 3 中Map支持的方式在当前的发布版中还未得到支持）。

Hibernate将集合分以下几类。

表 2.1. 集合语义

语义	Java实现类	注解
Bag 语义	java.util.List, java.util.Collection	@org.hibernate.annotations.CollectionOfElements 或 @OneToMany 或 @ManyToMany
List 语义	java.util.List	(@org.hibernate.annotations.CollectionOfElements 或 @OneToMany 或 @ManyToMany) 以及 @org.hibernate.annotations.IndexColumn
Set 语义	java.util.Set	@org.hibernate.annotations.CollectionOfElements 或 @OneToMany 或 @ManyToMany
Map 语义	java.util.Map	(@org.hibernate.annotations.CollectionOfElements 或 @OneToMany 或 @ManyToMany) 以及 (空 或 @org.hibernate.annotations.MapKey/MapKeyBy 支持真正的map), 或 @javax.persistence.MapKey

从上面可以明确地看到，没有@org.hibernate.annotations.IndexColumn 注解的java.util.List集合将被看作bag类。

EJB3规范不支持原始类型，核心类型，嵌入式对象的集合。但是Hibernate对此提供了支持（详情参考Hibernate Annotation Extensions）。

```
@Entity public class City {
    @OneToMany(mappedBy="city")
    @OrderBy("streetName")
    public List<Street> getStreets() {
        return streets;
    }
    ...
}

@Entity public class Street {
    public String getStreetName() {
        return streetName;
    }

    @ManyToOne
    public City getCity() {
        return city;
    }
    ...
}
```



```

}

@Entity
public class Software {
    @OneToMany(mappedBy="software")
    @MapKey(name="codeName")
    public Map<String, Version> getVersions() {
        return versions;
    }
    ...
}

@Entity
@Table(name="tbl_version")
public class Version {
    public String getCodeName() {...}

    @ManyToOne
    public Software getSoftware() { ... }
    ...
}

```

上面这个例子中, City 中包括了以streetName排序的Street的集合. 而Software中包括了以codeName作为 key 和以Version作为值的Map.

除非集合为generic类型, 否则你需要指定targetEntity. 这个注解属性接受的参数为目标实体的class.

2.2.5.3.2. 一对多 (One-to-many)

在属性级使用 @OneToMany注解可定义一对多关联. 一对多关联可以是双向关联.

2.2.5.3.2.1. 双向 (Bidirectional)

在EJB3规范中多对一这端几乎总是双向关联中的主体 (owner) 端, 而一对多这端的关联注解为 @OneToMany (mappedBy=...)

```

@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
        ...
    }
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
        ...
    }
}


```

Troop 通过troop 属性和Soldier建立了一对多的双向关联. 在mappedBy端不必也不能再定义任何物理映射

对于一对多的双向映射, 如果要一对多这一端维护关联关系, 你需要删除mappedBy元素并将多对一这端的 @JoinColumn的insertable和updatable设置为false. 很明显, 这种方案不会得到什么明显的优化, 而且还会增加一些附加的UPDATE语句.

```

@Entity
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") //we need to duplicate the physical information
    public Set<Soldier> getSoldiers() {
        ...
    }
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk", insertable=false, updatable=false) 
    public Troop getTroop() {
        ...
    }
}

```

2.2.5.3.2.2. 单向(Unidirectional)

通过在被拥有的实体端(owned entity)增加一个外键列来实现——**一对多单向关联**是很少见的,也是不推荐的. 我们强烈建议通过一个联接表(join table)来实现这种关联(下一节会对此进行解释). 可以通过@JoinColumn注解来描述这种单向关联关系.

```

@Entity
public class Customer implements Serializable {
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="CUST_ID")
    public Set<Ticket> getTickets() {
        ...
    }
}

@Entity
public class Ticket implements Serializable {
    ... //no bidir
}

```

Customer 通过 CUST_ID列和Ticket 建立了单向关联关系.

2.2.5.3.2.3. 通过关联表处理单向关联

通过联接表处理单向一对多关联是首选方式. 这种关联通过@JoinTable注解来进行描述.

```

@Entity
public class Trainer {
    @OneToMany
    @JoinTable(
        name="TrainedMonkeys",
        joinColumns = @JoinColumn( name="trainer_id"),
        inverseJoinColumns = @JoinColumn( name="monkey_id")
    )
    public Set<Monkey> getTrainedMonkeys() {
        ...
    }
}

@Entity
public class Monkey {

```

```
... //no bidir
}
```

上面这个例子中,Trainer通过 TrainedMonkeys表和 Monkey 建立了单向关联. 其中外键trainer_id关联到Trainer (joinColumns), 而外键monkey_id关联到 Monkey (inversejoinColumns).

2.2.5.3.2.4. 默认处理机制

通过联接表来建立单向一对多关联不需要描述任何物理映射. **表名**由以下三个部分组成:主表(owner table)表名+下划线+从表(the other side table)表名. **指向主表的外键名**: 主表表名+下划线+主表主键列名 **指向从表的外键名**: 主表所对应实体的属性名+下划线+从表主键列名 **指向从表的外键定义**为唯一约束,用来表示一对多的关联关系.

```
@Entity
public class Trainer {
    @OneToMany
    public Set<Tiger> getTrainedTigers() {
        ...
    }
}

@Entity
public class Tiger {
    ... //no bidir
}
```

上面这个例子中,Trainer和Tiger 通过联接表 Trainer_Tiger建立单向关联关系, 其中外键trainer_id关联到Trainer (主表表名, _(下划线), trainer id), 而外键trainedTigers_id关联到Tiger (属性名称, _(下划线), Tiger表的主键列名).

2.2.5.3.3. 多对多 (Many-to-many)

2.2.5.3.3.1. 定义

你可以通过@ManyToMany注解可定义的多对多关联. 同时,你也需要通过注解@JoinTable描述关联表和关联条件. 如果是双向关联,其中一段必须定义为owner,另一端必须定义为inverse(在对关联表进行更新操作时这一端将被忽略):

```
@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns=@JoinColumn(name="EMPER_ID"),
        inverseJoinColumns=@JoinColumn(name="EMPEE_ID")
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}
```

```
}
```

```
@Entity
public class Employee implements Serializable {
    @ManyToMany(
        cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        mappedBy = "employees",
        targetEntity = Employer.class
    )
    public Collection getEmployers() {
        return employers;
    }
}
```

至此, 我们已经展示了很多跟关联有关的声明定义以及属性细节. 下面我们将深入介绍@JoinTable注解, 该注解定义了联接表的表名, 联接列数组 (注解中定义数组的格式为{ A, B, C }), 以及inverse联接列数组. 后者是关联表中关联到Employee主键的列(the "other side").

正如前面所示, 被关联端不必也不能描述物理映射: 只需要一个简单的mappedBy参数, 该参数包含了主体端的属性名, 这样就绑定双方的关系.

2.2.5.3.3.2. 默认值

和其他许多注解一样, 在多对多关联中很多值是自动生成. 当双向多对多关联中没有定义任何物理映射时, Hibernate根据以下规则生成相应的值. 关联表名: 主表表名+_下划线+从表表名, 关联到主表的外键名: 主表名+_下划线+主表中的主键列名. 关联到从表的外键名: 主表中用于关联的属性名+_下划线+从表的主键列名. 以上规则对于双向一对多关联同样有效.

```
@Entity
public class Store {
    @ManyToMany(cascade = CascadeType.PERSIST)
    public Set<City> getImplantedIn() {
        ...
    }
}

@Entity
public class City {
    ... //no bidirectional relationship
}
```

上面这个例子中, Store_Table作为联接表. Store_id列是联接到Store表的外键. 而implantedIn_id列则联接到City表.

当双向多对多关联中没有定义任何物理映射时, Hibernate根据以下规则生成相应的值. 关联表名: :主表表名+_下划线+从表表名, 关联到主表的外键名: 从表用于关联的属性名+_下划线+主表中的主键列名. 关联到从表的外键名: 主表用于关联的属性名+_下划线+从表的主键列名. 以上规则对于双向一对多关联同样有效.

```

@Entity
public class Store {
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    public Set<Customer> getCustomers() {
        ...
    }
}

@Entity
public class Customer {
    @ManyToMany(mappedBy="customers")
    public Set<Store> getStores() {
        ...
    }
}

```

在上面这个例子中, Store_Customer作为联接表. stores_id列是联接到Store表的外键, 而customers_id列联接到City表.

2.2.5.4. 用cascading实现传播性持久化(Transitive persistence)

也许你已经注意到了cascade属性接受的值为CascadeType数组. 在EJB3中的cascade的概念和Hibernate中的传播性持久化以及cascade操作非常类似, 但是在语义上有细微的区别, 支持的cascade类型也有点区别:

- CascadeType.PERSIST: 如果一个实体是受管状态, 或者当persist()函数被调用时, 触发级联创建(create)操作
- CascadeType.MERGE: 如果一个实体是受管状态, 或者当merge()函数被调用时, 触发级联合并(merge)操作
- CascadeType.REMOVE: 当delete()函数被调用时, 触发级联删除(remove)操作
- CascadeType.REFRESH: 当refresh()函数被调用时, 触发级联更新(refresh)操作
- CascadeType.ALL: 以上全部

关于cascading, create/merge的语义请参考EJB3规范的6.3章节.

2.2.5.5. 关联关系获取

通过Hibernate你可以获得直接或者延迟获取关联实体的功能. **fetch参数**可以设置为FetchType.LAZY 或者 FetchType.EAGER. **EAGER通过outer join select直接获取关联的对象, 而LAZY(默认值)在第一次访问关联对象的时候才会触发相应的select操作.** EJBQL提供了fetch关键字, 该关键字可以在进行特殊查询的时候覆盖默认值. 这对于提高性能来说非常有效, 应该根据实际的用例来判断是否选择fetch关键字.

2.2.6. 映射复合主键与外键

组合主键使用一个可嵌入的类作为主键表示, 因此你需要使用@Id 和@Embeddable两个注解. 还有一种方式是使用@EmbeddedId注解. 注意所依赖的类必须实现 serializable以及实现equals()/hashCode()方法. 你也可以如Mapping identifier properties一章中描述的办法使用@IdClass注解.

```

@Entity
public class RegionalArticle implements Serializable {

    @Id
    public RegionalArticlePk getPk() { ... }
}

@Embeddable
public class RegionalArticlePk implements Serializable { ... }

```

或者

```

@Entity
public class RegionalArticle implements Serializable {

    @EmbeddedId
    public RegionalArticlePk getPk() { ... }
}

public class RegionalArticlePk implements Serializable { ... }

```

`@Embeddable` 注解默认继承了其所属实体的访问类型，除非显式使用了Hibernate的`@AccessType`注解(这个注解不是EJB3标准的一部分)。而`@JoinColumns`，即`@JoinColumn`数组，定义了关联的组合外键(如果不使用缺省值的话)。显式指明`referencedColumnName`是一个好的实践方式，否则，Hibernate认为你使用的列顺序和主键声明的顺序一致。

```

@Entity
public class Parent implements Serializable {
    @Id
    public ParentPk id;
    public int age;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumns ({
        @JoinColumn(name="parentCivility", referencedColumnName = "isMale"),
        @JoinColumn(name="parentLastName", referencedColumnName = "lastName"),
        @JoinColumn(name="parentFirstName", referencedColumnName = "firstName")
    })
    public Set<Child> children; //unidirectional
    ...
}

```

```

@Entity
public class Child implements Serializable {
    @Id @GeneratedValue
    public Integer id;

    @ManyToOne
    @JoinColumns ({
        @JoinColumn(name="parentCivility", referencedColumnName = "isMale"),
        @JoinColumn(name="parentLastName", referencedColumnName = "lastName"),
    })
}

```

```

        @JoinColumn(name="parentFirstName", referencedColumnName = "firstName")
    })
    public Parent parent; //unidirectional
}

```

```

@Embeddable
public class ParentPk implements Serializable {
    String firstName;
    String lastName;
    ...
}

```

注意上面的 `referencedColumnName` 显式使用方式。

2.2.7. 映射二级表(secondary tables)

使用类一级的 `@SecondaryTable` 或 `@SecondaryTables` 注解可以实现单个实体到多个表的映射。使用 `@Column` 或者 `@JoinColumn` 注解中的 `table` 参数可指定某个列所属的特定表。

```

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

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

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

    public String getName() {
        return name;
    }

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

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

```

在上面这个例子中, name保存在MainCat表中, storyPart1保存在Cat1表中, storyPart2保存在Cat2表中. Cat1表通过外键cat_id和MainCat表关联, Cat2表通过id列和MainCat表关联 (和MainCat的id列同名). 对storyPart2列还定义了唯一约束.

在JBoss EJB 3指南和Hibernate Annotations单元测试代码中还有更多的例子.

2.3. 映射查询

2.3.1. 映射EJBQL/HQL查询

使用注解还可以映射EJBQL/HQL查询. @NamedQuery 和@NamedQueries是可使用在类和包上的注解. 但是它们的定义在session factory/entity manager factory范围中是都可见的. 命名式查询通过它的名字和实际的查询字符串来定义.

```

javax.persistence.NamedQueries(
    @javax.persistence.NamedQuery(name="plane.getAll", query="select p from Plane p")
)
package org.hibernate.test.annotations.query;

...

@Entity
@NamedQuery(name="night.moreRecentThan", query="select n from Night n where n.date >= :date")
public class Night {
    ...
}

public class MyDao {
    doStuff() {
        Query q = s.getNamedQuery("night.moreRecentThan");
        q.setDate( "date", aMonthAgo );
        List results = q.list();
        ...
    }
    ...
}

```

还可以通过定义 QueryHint 数组的hints 属性为查询提供一些hint信息.

下面是目前可以使用的一些Hibernate hint:

表 2.2. Query hints

hint	description
org.hibernate.cacheable	查询是否与二级缓存交互(默认值为false)
org.hibernate.cacheRegion	设置缓存区名称 (默认为otherwise)
org.hibernate.timeout	查询超时设定

hint	description
org.hibernate.fetchSize	所获取的结果集(resultset)大小
org.hibernate.flushMode	本次查询所用的刷新模式
org.hibernate.cacheMode	本次查询所用的缓存模式
org.hibernate.readOnly	是否将本次查询所加载的实体设为只读(默认为false)
org.hibernate.comment	将查询注释添加入所生成的SQL

2.3.2. 映射本地化查询

你还可以映射本地化查询(也就是普通SQL查询). 不过这需要你使用@SqlResultSetMapping注解来描述SQL的resultset的结构(如果你打算定义多个结果集映射, 可是使用@SqlResultSetMappings). @SqlResultSetMapping和@NamedQuery, @SqlResultSetMapping一样, 可以定义在类和包一级. 但是@SqlResultSetMapping的作用域为应用级. 下面我们会看到, @NamedNativeQuery注解中resultSetMapping参数值为@SqlResultSetMapping的名字. 结果集映射定义了通过本地化查询返回值和实体的映射. 该实体中的每一个字段都绑定到SQL结果集中的某个列上. 该实体的所有字段包括子类的所有字段以及关联实体的外键列都必须在SQL查询中有对应的定义. 如果实体中的属性和SQL查询中的列名相同, 这种情况下可以不进行定义字段映射.

```
@NamedNativeQuery(name="night&area", query="select night.id nid, night.night_duration, "
+ " night.night_date, area.id aid, night.area_id, area.name "
+ "from Night night, Area area where night.area_id = area.id", resultSetMapping="joinMapping")
@SqlResultSetMapping(name="joinMapping", entities={
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Night.class, fields = {
        @FieldResult(name="id", column="nid"),
        @FieldResult(name="duration", column="night_duration"),
        @FieldResult(name="date", column="night_date"),
        @FieldResult(name="area", column="area_id"),
        discriminatorColumn="disc"
    }),
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class, fields = {
        @FieldResult(name="id", column="aid"),
        @FieldResult(name="name", column="name")
    })
})
}
```

在上面这个例子中, 名为night&area的查询和joinMapping结果集映射对应. 该映射返回两个实体, 分别为Night和Area, 其中每个属性都和一个列关联, 列名通过查询获取. 下面我们看一个隐式声明属性和列映射关系的例子.

```
@Entity
@SqlResultSetMapping(name="implicit", entities=@EntityResult(entityClass=org.hibernate.test.annotations.query.SpaceShip.class, fields={
    @FieldResult(name="name", column="name"),
    @FieldResult(name="model", column="model"),
    @FieldResult(name="speed", column="speed")
}))
@NamedNativeQuery(name="implicitSample", query="select * from SpaceShip", resultSetMapping="implicit")
public class SpaceShip {
    private String name;
    private String model;
    private double speed;
}
```

```

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

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

@Column(name="model_txt")
public String getModel() {
    return model;
}

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

public double getSpeed() {
    return speed;
}

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

```

在这个例子中, 我们只需要定义结果集映射中的实体成员. 属性和列名之间的映射借助实体中包含映射信息来完成. 在这个例子中, model属性绑定到model_txt列. 如果和相关实体的关联设计到组合主键, 那么应该使用@FieldResult注解来定义每个外键列. @FieldResult的名字由以下几部分组成: 定义这种关系的属性名字+“.”+主键名或主键列或主键属性.

```

@Entity
@SqlResultSetMapping(name="compositekey",
    entities=@EntityResult(entityClass=org.hibernate.test.annotations.query.SpaceShip.class,
        fields = {
            @FieldResult(name="name", column = "name"),
            @FieldResult(name="model", column = "model"),
            @FieldResult(name="speed", column = "speed"),
            @FieldResult(name="captain.firstname", column = "firstn"),
            @FieldResult(name="captain.lastname", column = "lastn"),
            @FieldResult(name="dimensions.length", column = "length"),
            @FieldResult(name="dimensions.width", column = "width")
        }),
    columns = { @ColumnResult(name = "surface"),
        @ColumnResult(name = "volume") } )

@NamedNativeQuery(name="compositekey",
    query="select name, model, speed, lname as lastn, fname as firstn, length, width, length * width as surface from SpaceShip",
    resultSetMapping="compositekey")
} )

public class SpaceShip {
    private String name;
    private String model;
    private double speed;
    private Captain captain;
    private Dimensions dimensions;

    @Id
    public String getName() {

```

```
        return name;
    }

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

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

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

    public String getModel() {
        return model;
    }

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

    public double getSpeed() {
        return speed;
    }

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

    public Dimensions getDimensions() {
        return dimensions;
    }

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

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

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

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

    @Id
```

```

public String getLastname() {
    return lastname;
}

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

```

注意

观察dimension属性你会发现Hibernate支持用“.”符号来表示嵌入式对象。EJB3实现不必支持这个特征,但是我们做到了:-)

如果查询返回的是单个实体,或者你打算使用系统默认的映射,这种情况下可以不使用resultSetMapping而是使用resultClass属性:

```

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

```

某些本地查询返回的是scalar值,例如报表查询。你可以通过@ColumnResult将其映射到@SqlResultSetMapping上。甚至还可以在同一个本地查询的结果中混合实体和scalar类型(不过这种情况比较少见)。

```

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

```

本地查询中还有另外一个hint属性: org.hibernate.callable。这个属性的布尔变量值表明这个查询是否是一个存储过程。

2.4. Hibernate独有的注解扩展

Hibernate 3.1 提供了多种附加的注解,这些注解可以与EJB3的实体混合/匹配使用。他们被设计成EJB3注解的自然扩展。

为了强化EJB3的能力,Hibernate提供了与其自身特性相吻合的特殊注解。org.hibernate.annotations包已包含了所有的这些注解扩展。

2.4.1. 实体

你可以在EJB3规范所能提供的能力之外,就Hibernate对实体所做的一些操作进行优化。

@org.hibernate.annotations.Entity 追加了可能需要的额外的元数据,而这些元数据超出了标准@Entity 中所定义的元数据。

- mutable: 此实体是否为可变的
- dynamicInsert: 用动态SQL新增
- dynamicUpdate: 用动态SQL更新

- `selectBeforeUpdate`: 指明Hibernate从不运行SQL UPDATE除非能确定对象的确已被修改
- `polymorphism`: (指出) 实体多态是 `PolymorphismType.IMPLICIT`(默认) 还是 `PolymorphismType.EXPLICIT`
- `persister`: 允许对默认持久实现(`persister implementation`)的覆盖
- `optimisticLock`: 乐观锁策略(`OptimisticLockType.VERSION`, `OptimisticLockType.NONE`, `OptimisticLockType.DIRTY`或`OptimisticLockType.ALL`)

注意

`@javax.persistence.Entity`仍是必选的(mandatory), `@org.hibernate.annotations.Entity`不是取代品.

以下是一些附加的Hibernate注解扩展:

`@org.hibernate.annotations.BatchSize` 允许你定义批量获取该实体的实例数量(如: `@BatchSize(size=4)`). 当加载一特定的实体时, Hibernate将加载在持久上下文中未经初始化的同类型实体, 直至批量数量(上限).

`@org.hibernate.annotations.Proxy` 定义了实体的延迟属性. `Lazy`(默认为true)定义了类是否为延迟(加载). `proxyClassName`是用来生成代理的接口(默认为该类本身).

`@org.hibernate.annotations.Where` 定义了当获取类实例时所用的SQL WHERE子句(该SQL WHERE子句为可选).

`@org.hibernate.annotations.Check` 定义了DDL语句中定义的合法性检查约束(该约束为可选).

`@OnDelete(action=OnDeleteAction.CASCADE)` 定义于被连接的子类(joined subclass): 在删除时使用SQL级连删除, 而非通常的Hibernate删除机制.

`@Table(name="tableName", indexes = { @Index(name="index1", columnNames={"column1", "column2"}) })` 在`tableName`表的列上创建定义好的索引. 该注解可以被应用于关键表或者是其他次要的表. `@Tables` 注解允许你在不同的表上应用索引. 此注解预期在使用 `@javax.persistence.Table`或 `@javax.persistence.SecondaryTable`的地方中出现.

注意

`@org.hibernate.annotations.Table` 是对 `@javax.persistence.Table`的补充而不是它的替代品. 特别是当你打算改变表名的默认值的时候, 你必须使用`@javax.persistence.Table`, 而不是 `@org.hibernate.annotations.Table`.

```
@Entity
@BatchSize(size=5)
@org.hibernate.annotations.Entity(
    selectBeforeUpdate = true,
    dynamicInsert = true, dynamicUpdate = true,
    optimisticLock = OptimisticLockType.ALL,
    polymorphism = PolymorphismType.EXPLICIT)
@Where(clause="1=1")
@org.hibernate.annotations.Table(name="Forest", indexes = { @Index(name="idx", columnNames = { "name", "length" }) })
public class Forest { ... }
```

```

@Entity
@Inheritance(
    strategy=InheritanceType.JOINED
)
public class Vegetable { ... }

@Entity
@OnDelete(action=OnDeleteAction.CASCADE)
public class Carrot extends Vegetable { ... }

```

2.4.2. 标识符

`@org.hibernate.annotations.GenericGenerator` 允许你定义一个Hibernate特定的id生成器.

```

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

@Id @GeneratedValue(generator="hibseq")
@GenericGenerator(name="hibseq", strategy = "seqhilo",
    parameters = {
        @Parameter(name="max_lo", value = "5"),
        @Parameter(name="sequence", value="heybabyhey")
    }
)
public Integer getId() {

```

`strategy`可以是Hibernate3生成器策略的简称, 或者是一个`IdentifierGenerator`实现的(带包路径的)全限定类名. 你可以通过`parameters`属性增加一些参数.

2.4.3. 属性

2.4.3.1. 访问类型

访问类型是根据`@Id`或`@EmbeddedId` 在实体继承层次中所处的位置推演而得的. 子实体(Sub-entities), 内嵌对象和被映射的父类均继承了根实体(root entity)的访问类型.

在Hibernate中, 你可以把访问类型覆盖成:

- 使用定制访问类型策略
- 优化类级或属性级的访问类型

为支持这种行为, Hibernate引入了`@AccessType`注解. 你可以对以下元素定义访问类型:

- 实体
- 父类
- 可内嵌的对象

- 属性

被注解元素的访问类型会被覆盖, 若覆盖是在类一级上, 则所有的属性继承访问类型. 对于根实体, 其访问类型会被认为是整个继承层次中的缺省设置(可在类或属性一级覆盖).

若访问类型被标以“property”, 则Hibernate会扫描getter方法的注解, 若访问类型被标以“field”, 则扫描字段的注解. 否则, 扫描标为@Id或@embeddedId的元素.

你可以覆盖某个属性(property)的访问类型, 但是受注解的元素将不受影响: 例如一个具有field访问类型的实体, (我们)可以将某个字段标注为 @AccessType(“property”), 则该字段的访问类型随之将成为property, 但是其他字段上依然需要携带注解.

若父类或可内嵌的对象没有被注解, 则使用根实体的访问类型(即使已经在非直系父类或可内嵌对象上定义了访问类型). 此时俄罗斯套娃(Russian doll)原理就不再适用. (译注: 俄罗斯套娃(матрешка 或 матрешка)是俄罗斯特产木制玩具, 一般由多个一样图案的空心木娃娃一个套一个组成, 最多可达十多个, 通常为圆柱形, 底部平坦可以直立.)

```
@Entity
public class Person implements Serializable {
    @Id @GeneratedValue //access type field
    Integer id;

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

@Embeddable
@AccessType("property") //override access type for all properties in Country
public class Country implements Serializable {
    private String iso2;
    private String name;

    public String getIso2() {
        return iso2;
    }

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

    @Column(name = "countryName")
    public String getName() {
        return name;
    }

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

2.4.3.2. 公式

有时候, 你想让数据库, 而非JVM, 来替你完成一些计算, 也可能想创建某种虚拟列. 你可以使用SQL片段(亦称为公式), 而不是将属性映射到(物理)列. 这种属性是只读的(属性值由公求得).

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

SQL片段可以是任意复杂的, 甚至可包含子查询.

2.4.3.3. 类型

`@org.hibernate.annotations.Type` 覆盖了Hibernate所用的默认类型: 这通常不是必须的, 因为类型可以由Hibernate正确推得. 关于Hibernate类型的详细信息, 请参考Hibernate使用手册.

`@org.hibernate.annotations.TypeDef` 和 `@org.hibernate.annotations.TypeDefs` 允许你来声明类型定义. 这些注解被置于类或包一级. 注意, 对session factory来说, 这些定义将是全局的(即使定义于类一级), 并且类型定义必须先于任何使用.

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

...
public class Forest {
    @Type(type="caster")
    public String getSmallText() {
        ...
    }
}
```

当使用组合的用户自定义类型时, 你必须自己表示列的定义. `@Columns`就是为了此目的而引入的.

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

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


2.4.3.4. 索引

通过在列属性(property)上使用@Index注解, 可以在特定列上定义索引, columnNames属性(attribute)将随之被忽略.

```
@Column(secondaryTable="Cat1")
@Index(name="storyindex")
public String getStoryPart1() {
    return storyPart1;
}
```

2.4.3.5. @Parent

在嵌入式对象内部, 你可以在那些指向该嵌入式对象所属元素的属性上定义该注解.

```
@Entity
public class Person {
    @Embeddable public Address address;
    ...
}

@Embeddable
public class Address {
    @Parent public Person owner;
    ...
}

person == person.address.owner
```

2.4.3.6. 生成的属性

某些属性可以在对数据库做插入或更新操作的时候生成. Hibernate能够处理这样的属性, 并触发一个后续的查询来读取这些属性.

```
@Entity
public class Antenna {
    @Id public Integer id;
    @Generated(GenerationTime.ALWAYS) @Column(insertable = false, updatable = false)
    public String longitude;

    @Generated(GenerationTime.INSERT) @Column(insertable = false)
    public String latitude;
}
```

你可以将属性注解为@Generated. 但是你要注意insertability和updatability不要和你选择的生成策略冲突. 如果选择了GenerationTime.INSERT, 该属性不能包含insertable列, 如果选择了GenerationTime.ALWAYS, 该属性不能包含insertable和updatable列.

@Version属性不可以为 @Generated(INSERT) (设计时), 只能是 NEVER或ALWAYS.

2.4.4. 继承

SINGLE_TABLE 是个功能强大的策略, 但有时, 特别对遗留系统而言, 是无法加入一个额外的辨别符列.

由此,Hibernate引入了辨别符公式(discriminator formula)的概念: `@DiscriminatorFormula`是 `@DiscriminatorColumn`的替代品,它使用SQL片段作为辨别符解决方案的公式(不需要有一个专门的字段)。

```
@Entity
@DiscriminatorFormula("case when forest_type is null then 0 else forest_type end")
public class Forest { ... }
```

2.4.5. 关于单个关联关系的注解

默认情况下,当预期的被关联元素不在数据库中(关乎关联列的错误id),致使Hiberante无法解决关联性问题时,Hibernate就会抛出异常。这对遗留schema和历经拙劣维护的schema而言,这或许有许多不便。此时,你可用 `@NotFound` 注解让Hibernate略过这样的元素而不是抛出异常。该注解可用于 `@OneToOne` (有外键)、`@ManyToOne`、`@OneToMany` 或 `@ManyToMany` 关联。

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

有时候删除某实体的时候需要触发数据库的级联删除。

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

上面这个例子中,Hibernate将生成一个数据库级的级联删除约束。

2.4.5.1. 延迟选项和获取模式

EJB3为延迟加载和获取模式提供了`fetch`选项,而Hibernate这方面提供了更丰富的选项集。为了更好的调整延迟加载和获取策略,Hibernate引入了一些附加的注解:

- `@LazyToOne`: 定义了 `@ManyToOne` 和 `@OneToOne` 关联的延迟选项。 `LazyToOneOption` 可以是 `PROXY` (例如:基于代理的延迟加载), `NO_PROXY` (例如:基于字节码增强的延迟加载 - 注意需要在构建期处理字节码) 和 `FALSE` (非延迟加载的关联)
- `@LazyCollection`: 定义了 `@ManyToMany`和 `@OneToMany` 关联的延迟选项。 `LazyCollectionOption` 可以是 `TRUE` (集合具有延迟性,只有在访问的时候才加载), `EXTRA` (集合具有延迟性,并且所有的操作都会尽量避免加载集合,对于一个巨大的集合特别有用,因为这样的集合中的元素没有必要全部加载) 和 `FALSE` (非延迟加载的关联)
- `@Fetch`: 定义了加载关联关系的获取策略。 `FetchMode` 可以是 `SELECT` (在需要加载关联的时候触发 `select`操作), `SUBSELECT` (只对集合有效,使用了子查询策略,详情参考Hibernate参考文档) 或 `JOIN`

(在加载主实体(owner entity)的时候使用SQL JOIN来加载关联关系). JOIN 将覆写任何延迟属性 (通过JOIN策略加载的关联将不再具有延迟性).

The Hibernate annotations overrides the EJB3 fetching options.

Hibernate注解覆写了EJB3提供的获取(fetch)选项.

表 2.3. 延迟和获取选项的等效注解

Annotations	Lazy	Fetch
@One @Many @ManyToOne (fetch=FetchType.LAZY)	@Lazy (TRUE)	@Fetch (SELECT)
@One @Many @ManyToOne (fetch=FetchType.EAGER)	@Lazy (FALSE)	@Fetch (JOIN)
@OneToMany (fetch=FetchType.LAZY)	@LazyCollection (TRUE)	@Fetch (SELECT)
@OneToMany (fetch=FetchType.EAGER)	@LazyCollection (FALSE)	@Fetch (JOIN)

2.4.6. 关于集合类型的注解

2.4.6.1. 参数注解

以下是可能的设置方式

- 用@BatchSizebatch设置集合的batch大小
- 用@Where注解设置Where子句
- 用注解@Check来设置check子句
- 用注解@OrderBy来设置SQL的order by子句
- 利用@OnDelete(action=OnDeleteAction.CASCADE) 注解设置级连删除策略

你也可以利用@Sort注解定义一个排序比较器(sort comparator), 表明希望的比较器类型, 无序、自然顺序或自定义排序, 三者择一. 若你想用你自己实现的comparator, 你还需要利用comparator属性(attribute)指明实现类.

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Sort(type = SortType.COMPARATOR, comparator = TicketComparator.class)
@Where(clause="1=1")
@OnDelete(action=OnDeleteAction.CASCADE)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

关于这些注解更详细的信息, 请参阅此前的描述.

2.4.6.2. 更多的集合类型

比EJB3更胜一筹的是, Hibernate Annotations支持真正的 List和Array. 映射集合的方式和以前完全一样, 只不过要新增@IndexColumn注解. 该注解允许你指明存放索引值的字段. 你还可以定义代表数据库中首个元素的索引值(亦称为索引基数). 常见取值为0或1.

```
@OneToMany(cascade = CascadeType.ALL)
@IndexColumn(name = "drawer_position", base=1)
public List<Drawer> getDrawers() {
    return drawers;
}
```

注意

假如你忘了设置@IndexColumn, Hibernate会采用包(bag)语义(译注: 即允许重复元素的无序集合).

Hibernate注解支持true Map映射, 如果没有设置@javax.persistence.MapKey, hibernate将key元素或嵌入式对象直接映射到他们所属的列. 要覆写默认的列, 可以使用以下注解: @org.hibernate.annotations.MapKey适用的key为基本类型或者嵌入式对象, @org.hibernate.annotations.MapKey适用的key为实体.

Hibernate Annotations还支持核心类型集合(Integer, String, Enums,)、可内嵌对象的集合, 甚至基本类型数组. 这就是所谓的元素集合.

元素集合可用@CollectionOfElements来注解(作为@OneToMany的替代). 为了定义集合表(译注: 即存放集合元素的表, 与下面提到的主表对应), 要在关联属性上使用@JoinTable注解, joinColumns定义了介乎实体主表与集合表之间的连接字段(inverseJoincolumn是无效的且其值应为空). 对于核心类型的集合或基本类型数组, 你可以在关联属性上用@Column来覆盖存放元素值的字段的定义. 你还可以用@AttributeOverride来覆盖存放可内嵌对象的字段的定义. 要访问集合元素, 需要将该注解的name属性值设置为"element"("element"用于核心类型, 而"element.serial"用于嵌入式对象的serial属性). 要访问集合的index/key, 则将该注解的name属性值设置为"key".

```
@Entity
public class Boy {
    private Integer id;
    private Set<String> nickNames = new HashSet<String>();
    private int[] favoriteNumbers;
    private Set<Toy> favoriteToys = new HashSet<Toy>();
    private Set<Character> characters = new HashSet<Character>();

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

    @CollectionOfElements
    public Set<String> getNickNames() {
        return nickNames;
    }

    @CollectionOfElements
    @JoinTable(
        table=@Table(name="BoyFavoriteNumbers"),
        joinColumns = @JoinColumn(name="BoyId")
    )
    @Column(name="favoriteNumber", nullable=false)
    @IndexColumn(name="nbr_index")
    public int[] getFavoriteNumbers() {
```

```

        return favoriteNumbers;
    }

    @CollectionOfElements
    @AttributeOverride( name="element.serial", column=@Column(name="serial_nbr") )
    public Set<Toy> getFavoriteToys() {
        return favoriteToys;
    }

    @CollectionOfElements
    public Set<Character> getCharacters() {
        return characters;
    }
    ...
}

public enum Character {
    GENTLE,
    NORMAL,
    AGGRESSIVE,
    ATTENTIVE,
    VIOLENT,
    CRAFTY
}

@Embeddable
public class Toy {
    public String name;
    public String serial;
    public Boy owner;

    public String getName() {
        return name;
    }

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

    public String getSerial() {
        return serial;
    }

    public void setSerial(String serial) {
        this.serial = serial;
    }

    @Parent
    public Boy getOwner() {
        return owner;
    }

    public void setOwner(Boy owner) {
        this.owner = owner;
    }

    public boolean equals(Object o) {
        if ( this == o ) return true;
        if ( o == null || getClass() != o.getClass() ) return false;

```

```

        final Toy toy = (Toy) o;

        if ( !name.equals( toy.name ) ) return false;
        if ( !serial.equals( toy.serial ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = name.hashCode();
        result = 29 * result + serial.hashCode();
        return result;
    }
}

```

在嵌入式对象的集合中, 可以使用 `@Parent` 注解嵌入式对象的某属性. 该属性指向该嵌入式对象所属的集合实体.

注意

旧版的Hibernate Annotations用`@OneToMany`来标识集合元素. 由于语义矛盾, 我们引入了`@CollectionOfElements`注解. 用`@OneToMany`来标识集合元素的这种旧有方式目前尚有效, 但是不推荐使用, 而且在以后的发布版本中不再支持这种方式.

2.4.7. 缓存

为了优化数据库访问, 你可以激活所谓的Hibernate二级缓存. 该缓存是可以按每个实体和集合进行配置的.

`@org.hibernate.annotations.Cache`定义了缓存策略及给定的二级缓存的范围. 此注解适用于根实体(非子实体), 还有集合.

```

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

```

```

@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() {
    return tickets;
}

```

```

@Cache(
    CacheConcurrencyStrategy usage();           (1)
    String region() default "";                 (2)
    String include() default "all";             (3)
)

```

- (1) `usage`: 给定缓存的并发策略(NONE, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL)
- (2) `region` (可选的): 缓存范围(默认为类的全限定类名或是集合的全限定角色名)

- (3) include (可选的): 值为all时包括了所有的属性(property), 为non-lazy时仅含非延迟属性(默认值为all)

2.4.8. 过滤器

Hibernate具有数据过滤器的概念, 可在运行期应用于一个给定的session. 过滤器需要事先定义好.

@org.hibernate.annotations.FilterDef 或 @FilterDefs 定义过滤器声明, 为同名过滤器所用. 过滤器声明带有一个name() 和一个parameters() 数组, @ParamDef包含name和type, 你还可以为给定的@filterDef定义一个defaultCondition() 参数, 当@Filter中没有任何定义时, 可使用该参数定义缺省条件. @FilterDef (s) 可以在类或包一级进行定义.

现在我们来定义应用于实体或集合加载时的SQL过滤器子句. 我们使用@Filter, 并将其置于实体或集合元素上.

```
@Entity
@FilterDef(name="minLength", parameters=@ParamDef( name="minLength", type="integer" ))
@Filters( {
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length"),
    @Filter(name="minLength", condition=":minLength <= length")
} )
public class Forest { ... }
```

2.4.9. 查询

由于 Hibernate 引入了 @org.hibernate.annotations.NamedQuery, @org.hibernate.annotations.NamedQueries, @org.hibernate.annotations.NamedNativeQuery 和 @org.hibernate.annotations.NamedNativeQueries 命名式查询, 因此Hibernate在命名式查询上比EJB3规范中所定义的命名式查询提供了更多的特性. 他们在标准版中添加了可作为替代品的一些属性(attributes):

- flushMode: 定义查询的刷新模式(Always, Auto, Commit或Manual)
- cacheable: 查询该不该被缓存
- cacheRegion: 若查询已被缓存时所用缓存的范围
- fetchSize: 针对该查询的JDBC statement单次获取记录的数目
- timeout: 查询超时
- callable: 仅用于本地化查询(native query), 对于存储过程设为true
- comment: 一旦激活注释功能, 我们会在向数据库交送查询请求时看到注释
- cacheMode: 缓存交护模式(get, ignore, normal, 或refresh)
- readOnly: 不管是否从查询获取元素都是在只读模式下

注意, EJB3已公开的最终草案中引入了@QueryHint的概念, 这可能是定义hints更好的方法.

第 3 章 通过XML覆写元数据

在EJB3中元数据的主要目标是使用注释,但是EJB3规范也提供通过XML部署文件来覆写或者替换元数据注释. 在当前的发布版本仅仅支持EJB3注释的覆写,如果你想使用Hibernate特有的一些实体注释,你有两种选择:一,只使用注释;二,使用原来的hbm 映射文件.你当然还是可以同时使用注释实体和hbm XML映射文件的实体.

在测试套件中有一些附加的XML文件的样例.

3.1. 原则

XML部署文件结构被设计为直接映射注释结构,所以如果你知道注释的结构,那么使用XML语法是很简单的.

你可以定义一个或者多个XML文件来描述你的元数据,这些文件会被覆写引擎合并(merged).

3.1.1. 全局级别的元数据

你可以使用XML文件来定义全局元数据,对每一个部署文件你不能定义多于一个的元数据.

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0">

  <persistence-unit-metadata>
    <xml-mapping-metadata-complete/>
    <persistence-unit-defaults>
      <schema>myschema</schema>
      <catalog>mycatalog</catalog>
      <cascade-persist/>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
```

xml-mapping-metadata-complete 意味着所有的实体, mapped-superclasses和嵌套的元数据应该从XML文件中启用(忽略注释).

schema / catalog 将覆写所有在元数据中默认定义的schema 和 catalog(包括XML和注释).

cascade-persist 意味着所有注释作为一个 cascade type 都是PERSIST的. 我们推荐你不要使用该特性.

3.1.2. 实体级别的元数据

你也可以在一个给定的实体上定义或者覆写元数据

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  (1)
```



```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
version="1.0">

  <package>org.hibernate.test.annotations.reflection</package> (2)
  <entity class="Administration" access="PROPERTY" metadata-complete="true"> (3)
    <table name="tbl_admin"> (4)
      <unique-constraint>
        <column-name>firstname</column-name>
        <column-name>lastname</column-name>
      </unique-constraint>
    </table>
    <secondary-table name="admin2"> (5)
      <primary-key-join-column name="admin_id" referenced-column-name="id"/>
      <unique-constraint>
        <column-name>address</column-name>
      </unique-constraint>
    </secondary-table>
    <id-class class="SocialSecurityNumber"/> (6)
    <inheritance strategy="JOINED"/> (7)
    <sequence-generator name="seqhilo" sequence-name="seqhilo"/> (8)
    <table-generator name="table" table="tablehilo"/> (9)
    ...
  </entity>

  <entity class="PostalAdministration">
    <primary-key-join-column name="id"/> (10)
    ...
  </entity>
</entity-mappings>

```

- (1) `entity-mappings:entity-mappings` 是所有XML文件的根元素. 你必须定义XML Schema, 该文件包含在 `hibernate-annotations.jar` 中, 使用Hibernate Annotations 不需要访问网络.
- (2) `package` (可选的): 作为默认的package用于在一个给定的部署描述文件中所有没有限定的类.
- (3) `entity`: 描述一个实体.

`metadata-complete` 定义对于该元素是否全部使用元数据(换句话说就是, 如果注释出现在类级别应该考虑或者忽略).

一个实体不得不有一个 `class` 属性来引用 元数据所应用的类.

通过`name`属性你可以覆写实体的名字, 如果没有定义并且`@Entity.name`出现了的话, 那么就使用该注释(假如`metadata complete` 没有被设置).

对于`metadata complete` (参考下面)元素, 你可以定义一个 `access` (FIELD 或者 PROPERTY(默认值)), 对于非`metadata complete` 元素, 使用注释的`access type`.

- (4) `table`: 你可以声明`table` 属性(`name`, `schema`, `catalog`), 如果没有定义, 将使用Java注释.

就象例子中所示的那样你可以定义一个或者多个`unique constraints`

- (5) `secondary-table`: 定义一个 `secondary-table`, 除了你可以通过`primary-key-join-column` 元素定义 `primary key` / `foreign key` 列以外是和一般的`table`一样的. 在非`metadata complete`下, `annotation secondary tables` 仅仅在没有`secondary-table` 定义的情况下使用, 否则 注释将被忽略.
- (6) `id-class`: 和`@IdClass`一样定义一个`id class`.
- (7) `inheritance`: 定义继承策略(`JOINED`, `TABLE_PER_CLASS`, `SINGLE_TABLE`), 仅仅在根实体级别可以使用.

- (8) sequence-generator: 定义一个序列产生器.
- (9) table-generator: 定义一个table generator
- (10) primary-key-join-column: 当 JOINED 继承策略使用时, 为sub entities定义一个 primary key join column.

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_1_0.xsd"
  version="1.0">

  <package>org.hibernate.test.annotations.reflection</package>
  <entity class="Music" access="PROPERTY" metadata-complete="true">
    <discriminator-value>Generic</discriminator-value>
    <discriminator-column length="34"/>
    ...
  </entity>

  <entity class="PostalAdministration">
    <primary-key-join-column name="id"/>
    <named-query name="adminById">
      <query>select m from Administration m where m.id = :id</query>
      <hint name="org.hibernate.timeout" value="200"/>
    </named-query>
    <named-native-query name="allAdmin" result-set-mapping="adminrs">
      <query>select *, count(taxpayer_id) as taxPayerNumber
      from Administration, TaxPayer
      where taxpayer_admin_id = admin_id group by ...</query>
      <hint name="org.hibernate.timeout" value="200"/>
    </named-native-query>
    <sql-result-set-mapping name="adminrs">
      <entity-result entity-class="Administration">
        <field-result name="name" column="fld_name"/>
      </entity-result>
      <column-result name="taxPayerNumber"/>
    </sql-result-set-mapping>
    <attribute-override name="ground">
      <column name="fld_ground" unique="true" scale="2"/>
    </attribute-override>
    <association-override name="referer">
      <join-column name="referer_id" referenced-column-name="id"/>
    </association-override>
    ...
  </entity>
</entity-mappings>
```

- (1) discriminator-value / discriminator-column: 当SINGLE_TABLE继承策略使用时, 定义鉴别器值 和 保存该值的列.
- (2) named-query: 定义命名查询和一些相关的可能的线索. 该定义附加在注释的定义中, 如果两个都定义了相同的名字, 那么XML将优先考虑.
- (3) named-native-query: 定义一个命名本地查询 和他的 sql result set 映射. 作为另外一种选择, 你可以定义result-class. 这些定义附加在注释的定义中. 如果两个定义了同样的名字, XML文件优先考虑.
- (4) sql-result-set-mapping: 描述了 result set mapping 的结构. 你可以定义 实体和列映射. 这些定义附加在注释的定义中, 如果定义了同样的名字, XML文件优先考虑.

- (5) `attribute-override` / `association-override`: 定义一列或者join column overriding. 该overriding附加在注释的定义中.

一些应用于 `<embeddable>` 和 `<mapped-superclass>`.

3.1.3. 属性级别的元数据

你当然可以定义XML来覆写属性. 如果`metadata complete`给定义了, 那么附加的属性(如: 在Java 级别的)将被忽略. 另外, 一旦你开始覆写一个属性, 在该属性上的所有注释都会被忽略. 所有属性级别的元数据应用于`entity/attributes`, `mapped-superclass/attributes` 或 `embeddable/attributes`.

```
<attributes>
  <id name="id">
    <column name="fld_id"/>
    <generated-value generator="generator" strategy="SEQUENCE"/>
    <temporal>DATE</temporal>
    <sequence-generator name="generator" sequence-name="seq"/>
  </id>
  <version name="version"/>
  <embedded name="embeddedObject">
    <attribute-override name="subproperty">
      <column name="my_column"/>
    </attribute-override>
  </embedded>
  <basic name="status" optional="false">
    <enumerated>STRING</enumerated>
  </basic>
  <basic name="serial" optional="true">
    <column name="serialbytes"/>
    <lob/>
  </basic>
  <basic name="terminusTime" fetch="LAZY">
    <temporal>TIMESTAMP</temporal>
  </basic>
</attributes>
```

通过 `id`, `embedded-id`, `version`, `embedded` 和 `basic`你可以覆写一个属性, 这些元素中的每一个元素都有相应的subelements: `lob`, `temporal`, `enumerated`, `column`.

3.1.4. 关联级别的元数据

你可以定义XML覆写关联注释. 所有的关联级别的元数据作用于 `entity/attributes`, `mapped-superclass/attributes` 或 `embeddable/attributes`.

```
<attributes>
  <one-to-many name="players" fetch="EAGER">
    <map-key name="name"/>
    <join-column name="driver"/>
    <join-column name="number"/>
  </one-to-many>
  <many-to-many name="roads" target-entity="Administration">
    <order-by>maxSpeed</order-by>
    <join-table name="bus_road">
      <join-column name="driver"/>
      <join-column name="number"/>
      <inverse-join-column name="road_id"/>
    </join-table>
  </many-to-many>
</attributes>
```

```
        <unique-constraint>
            <column-name>driver</column-name>
            <column-name>number</column-name>
        </unique-constraint>
    </join-table>
</many-to-many>
    <many-to-many name="allTimeDrivers" mapped-by="drivenBuses">
</attributes>
```

通过 one-to-many, one-to-one, many-to-one, 和 many-to-many. 你可以重写一个关联关系. 这些元素中的每一个都有相应的 subelements. join-table (可以有 join-column 和 inverse-join-column), join-column, map-key, 和 order-by. mapped-by 和 target-entity 当它们有意义的时候可以定义属性. 再一次强调 该结构映射于注释的结构. 在描述注释的一章中 你可以找到所有的语义信息.

第 4 章 Hibernate验证器

注解是一种为领域模型(domain model)指定不变约束的简洁而幽雅的方法。例如，你能 表示一个属性永远不为null，一个帐户余额一定是正值，等等。这些域模型约束通过为bean中的属性添加 注解来加以声明。随后一个验证器(validator)会读取并检查这些约束。验证机制可以执行于应用程序中的 不同层（表现层、数据访问层），而不必复述任何（前述）这些规则。Hibernate验证器正为这一目的而设计的。

Hibernate验证器工作在两个层次上。第一层，它能检查内存中一个类的实例是否违反约束。 第二层，它能将约束应用于Hibernate元模型上，并将它们融入生成的数据库schema中。

每个约束注解（constraint annotation）和一个验证器实现关联，该验证器负责检查位于实体实例上的约束。 一个验证器也能(可选地)将约束应用于Hibernate元模型上，让Hibernate生成表示这一约束的DDL。使用合适的事件监听器，你能 让Hibernate在插入和更新时执行检查操作。Hibernate验证器并不局限于同Hibernate一起使用。 你能在你应用程序的任何地方方便地使用它。

在运行时检查实例时，Hibernate验证器返回违反约束的信息， 这些信息以一个InvalidValue数组的形式返回。除了众多其他信息外，InvalidValue包含了一个错误描述消 息，该信息可以内嵌与注解相捆绑的参数值（例如长度限制），以及能被提取至ResourceBundle的消息字符串。

4.1. 约束

4.1.1. 什么是约束？

约束通过注解表示。一个约束通常有一些用来参数化约束限制的属性。约束应用于带注解的元素。

4.1.2. 内建约束

Hibernate验证器有些内建约束，这些约束覆盖了大多数的基本数据检查。随后我们会看到， 你不必受制于这些内置约束，因为一分钟内就可以写出你自己的约束。

表 4.1. 内建约束

注解	应用目标	运行时检查	Hibernate元数据影响
@Length(min=, max=)	属性(String)	检查字符串长度是否符合范围	列长度会被设到最大值
@Max(value=)	属性（以numeric或者string类型来表示一个数字）	检查值是否小于或等于最大值	对列增加一个检查约束
@Min(value=)	属性（以numeric或者string类型来表示一个数字）	检查值是否大于或等于最小值	对列增加一个检查约束
@NotNull	属性	检查值是否非空(not null)	列不为空

注解	应用目标	运行时检查	Hibernate元数据影响
@Past	属性 (date或calendar)	检查日期是否是过去时	对列增加一个检查约束
@Future	属性 (date 或 calendar)	检查日期是否是将来时	无
@Pattern(regex="regex", flag=)	属性 (string)	检查属性是否与给定匹配标志的正则表达式相匹配 (见 java.util.regex.Pattern)	无
@Range(min=, max=)	属性 (以 numeric 或者 string类型来表示一个数字)	检查值是否在最小和最大值之间(包括临界值)	对列增加一个检查约束
@Size(min=, max=)	属性 (array, collection, map)	检查元素大小是否在最小和最大值之间(包括临界值)	无
@AssertFalse	属性	检查方法的演算结果是否为false(对以代码方式而不是注解表示的约束很有用)	无
@AssertTrue	属性	检查方法的演算结果是否为true(对以代码方式而不是注解表示的约束很有用)	无
@Valid	属性 (object)	对关联对象递归的进行验证。如果对象是集合或数组，就递归地验证其元素。如果对象是Map，则递归验证其值元素。	无
@Email	属性 (String)	检查字符串是否符合有效的email地址规范。	无

4.1.3. 错误信息

Hibernate验证器提供了一组默认的错误提示信息，它们被翻译成多种语言(如果你的语言不在其中，请给我们寄一个补丁)。你可以在org.hibernate.validator.resources.DefaultValidatorMessages.properties 之外创建ValidatorMessages.properties或ValidatorMessages_loc.properties 文件并改变相应的键值，籍此覆盖那些(默认)信息。你甚至可以在写自己的验证器 注解时添加你自己的附加消息集。

或者你可以以编程方式检查bean的验证规则并提供相应的ResourceBundle。

4.1.4. 编写你自己的约束

扩展内建约束集是极其方便的。任何约束都包括两部分：约束描述符(注解) 和约束验证器(实现类)。下面是一个简单的用户定义描述符：

```
@ValidatorClass(CapitalizedValidator.class)
@Target(METHOD)
@Retention(RUNTIME)
@Documented
public @interface Capitalized {
    CapitalizeType type() default Capitalize.FIRST;
    String message() default "has incorrect capitalization";
}
```

type参数描述属性应该如何被大写。这是一个完全依赖于注解业务(逻辑)的用户 参数。

message是用于描述约束违规的默认字符串，它是强制要求的。你可以采取硬编码的方式， 或者通过Java ResourceBundle机制将message的部分/全部内容提取至外部文件。一旦发现message中 {parameter} 字符串， 就会在 {parameter} 这个位置注入相应的参数值(在我们的例子里 Capitalization is not {type}会生成 Capitalization is not FIRST)， 可以将message对应的整个字符串提取至外部文件ValidatorMessages.properties，这也是一种良好实践。 见Error messages。

```
@ValidatorClass(CapitalizedValidator.class)
@Target(METHOD)
@Retention(RUNTIME)
@Documented
public @interface Capitalized {
    CapitalizeType type() default Capitalize.FIRST;
    String message() default "{validator.capitalized}";
}

...
#in ValidatorMessages.properties
validator.capitalized=Capitalization is not {type}
```

如你所见 {} 符号是递归的。

为了将一个描述符连接到它的验证器实现，我们使用@ValidatorClass 元注解。验证器类参数必须指定一个实现了Validator<ConstraintAnnotation> 的类。

我们现在要实现验证器(也就是实现规则检查)。一个验证器实现能检查一个属性的值(实现PropertyConstraint)，并且/或者可以修改hibernate映射元数据(实现PersistentClassConstraint)，籍此表示数据库级的约束。

```
public class CapitalizedValidator
    implements Validator<Capitalized>, PropertyConstraint {
    private CapitalizeType type;

    //part of the Validator<Annotation> contract,
    //allows to get and use the annotation values
    public void initialize(Capitalized parameters) {
        type = parameters.type();
    }

    //part of the property constraint contract
    public boolean isValid(Object value) {
        if (value==null) return true;
        if ( !(value instanceof String) ) return false;
```

```
String string = (String) value;
if (type == CapitalizeType.ALL) {
    return string.equals( string.toUpperCase() );
}
else {
    String first = string.substring(0,1);
    return first.equals( first.toUpperCase());
}
}
```

如果违反约束，`isValid()`方法将返回`false`。更多例子请参考内建验证器实现。

至此我们只看到属性级的验证，你还可以写一个Bean级别的验证注解。Bean自身会被传递给验证器，而不是bean的属性实例。只要对bean自身进行注解即可激活验证检查。在单元测试套件中还可以找到一个小例子。

4. 1. 5. 注解你的领域模型

既然你现在已经熟悉注解了，那么对语法也应该很清楚了。

```
public class Address {
    private String line1;
    private String line2;
    private String zip;
    private String state;
    private String country;
    private long id;

    // a not null string of 20 characters maximum
    @Length(max=20)
    @NotNull
    public String getCountry() {
        return country;
    }

    // a non null string
    @NotNull
    public String getLine1() {
        return line1;
    }

    //no constraint
    public String getLine2() {
        return line2;
    }

    // a not null string of 3 characters maximum
    @Length(max=3) @NotNull
    public String getState() {
        return state;
    }

    // a not null numeric string of 5 characters maximum
    // if the string is longer, the message will
    //be searched in the resource bundle at key 'long'
    @Length(max=5, message="{long}")
    @Pattern(regex="[0-9]+")
```



```

    @NotNull
    public String getZip() {
        return zip;
    }

    // should always be true
    @AssertTrue
    public boolean isValid() {
        return true;
    }

    // a numeric between 1 and 2000
    @Id @Min(1)
    @Range(max=2000)
    public long getId() {
        return id;
    }
}

```

上面的例子只展示了公共属性验证，你还可以对任何可见度的字段(field)进行注解。

```

@MyBeanConstraint(max=45)
public class Dog {
    @AssertTrue private boolean isMale;
    @NotNull protected String getName() { ... };
    ...
}

```

你可以对接口进行注解。Hibernate验证器会检查给定bean所扩展或实现的所有父类和接口，籍以读取相应的验证器注解(信息)。

```

public interface Named {
    @NotNull String getName();
    ...
}

public class Dog implements Named {

    @AssertTrue private boolean isMale;

    public String getName() { ... };

}

```

在验证Dog bean时会检查name属性的有效性(不为null)。

4.2. 使用验证器框架

Hibernate验证器旨在实现多层数据验证，我们在一处表示约束(带注解的域模型)，然后将其运用于应用程序的不同层。

4.2.1. 数据库schema层次验证

无须额外手续，Hibernate Annotations会自动将 you 为实体定义的约束翻译为映射元数据。例如，如果

你的实体 的一个属性注解为@NotNull，在Hibernate生成的DDL schema中这列会被定义为 not null。

4.2.2. Hibernate基于事件的验证

Hibernate验证器有两个内建Hibernate事件监听器。当一个PreInsertEvent 或PreUpdateEvent发生时，监听器会验证该实体实例的所有约束，如有违反会抛出一个异常。基本上，在Hibernate执行任何插入和更新前对象会被检查。这是激活验证过程的最便捷最简单的方法。当遇到约束 违规时，事件会引发一个运行时InvalidStateException，该异常包含一个描述每个错误的 InvalidValue数组。

```
<hibernate-configuration>
...
<event type="pre-update">
  <listener
    class="org.hibernate.validator.event.ValidatePreUpdateEventListener"/>
</event>
<event type="pre-insert">
  <listener
    class="org.hibernate.validator.event.ValidatePreInsertEventListener"/>
</event>
</hibernate-configuration>
```

注意

在使用Hibernate Entity Manager时，Validation框架会被自动激活。如果bean不带验证注解，就不会有性能损失。

4.2.3. 程序级验证

Hibernate验证器能应用于你应用程序代码中的任何地方。

```
ClassValidator personValidator = new ClassValidator( Person.class );
ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("messages", Locale.ENGLISH)

InvalidValue[] validationMessages = addressValidator.getInvalidValues(address);
```

头两行为执行类检查而准备Hibernate验证器。第一行依赖于嵌入在Hibernate验证器内的错误 消息(见Error messages)，第二行为这些消息准备资源包。这些代码只执行一次，并将验证器进行缓存处理，这种方式是一种良好实践。

第三行真正验证了Address实例并返回一个InvalidValue数组。 你的应用程序逻辑随后可以对错误做出响应。

除了针对整个bean你还可以对某个特定属性进行检查。这对于一个属性一个属性的用户交互情形或许是有用的。

```
ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("messages", Locale.ENGLISH)

//only get city property invalid values
InvalidValue[] validationMessages = addressValidator.getInvalidValues(address, "city");

//only get potential city property invalid values
InvalidValue[] validationMessages = addressValidator.getPotentialInvalidValues("city", "Paris")
```

4.2.4. 验证信息

作为一个验证信息的载体，hibernate提供了一个InvalidValue数组。每个InvalidValue有一组，这些方法分别描述相应的个体问题。

`getBeanClass()` 获取失败的bean类型。

`getBean()` 获取验证失败的实例(如果有的话，当使用 `getPotentialInvalidValues()` 时则不会取到)

`getValue()` 获取验证失败的值

`getMessage()` 获取合适的国际化错误消息

`getRootBean()` 获取产生问题的根bean实例（在与 `@Valid` 连用时很有用），如用 `getPotentialInvalidValues()` 则返回null。

`getPropertyPath()` 获取“问题”属性从根bean开始的带点的路径

第 5 章 Hibernate与Lucene集成

Lucene是一个高性能的java搜索引擎库，可以从 Apache软件基金组织获取。 Hibernate Annotations 包括一个注解包，它允许把任何域模型对象标记为可索引的，并且对任何经由Hibernate进行持续化的实例，Hibernate 都会为之维护一个对应的Lucene索引。

5.1. 使用Lucene为实体建立索引

5.1.1. 注解领域模型

首先，必须将一个持久类声明为 @Indexed：

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...
}
```

属性index是告诉Hibernate， Lucene索引信息所在的位置（你文件系统的某个目录）。如果你想为所有的Lucene索引定义一个根目录，你可以在配置文件中用属性hibernate.lucene.index_dir进行配置。

Lucene索引包括四种字段：keyword 字段，text 字段，unstored字段和unindexed字段。 Hibernate 注解提供了将实体属性标记为前三种被索引字段的注解。

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...

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

    @Text(name="Abstract")
    public String getSummary() { return summary; }

    @Lob
    @Unstored
    public String getText() { return text; }
}
```

这些注解定义了一个带有三个字段的索引：Id, Abstract 和 Text.

注意:你必须在你的实体类的标志属性上指定 @Keyword(id=true) .

用于对元素建立索引的分析器类是可以通过hibernate.lucene.analyzer属性进行配置的。 如果没有定义，则把 org.apache.lucene.analysis.standard.StandardAnalyzer作为缺省。

5.1.2. 启用自动索引

我们激活用于监听三类Hibernate事件的 `LuceneEventListener`， 这些事件会在变更被提交至数据库后产生。

```
<hibernate-configuration>
...
<event type="post-commit-update"
  <listener
    class="org.hibernate.lucene.event.LuceneEventListener"/>
</event>
<event type="post-commit-insert"
  <listener
    class="org.hibernate.lucene.event.LuceneEventListener"/>
</event>
<event type="post-commit-delete"
  <listener
    class="org.hibernate.lucene.event.LuceneEventListener"/>
</event>
</hibernate-configuration>
```

附录 A. 术语表

Redsaga 的 wiki 上维护了本文翻译过程中所参照的中英文对照的术语表, 地址:
<http://wiki.redsaga.com/confluence/display/HART/glossary>.