1. SessionFactory是一个线程安全的、不可更改的，表示数据库和应用domain映射关系。SessionFactory的创建代价非常高，对于给定的数据库，相应的SessionFactory只应该有一个。

2. Session: 一个单线程，短存活的实例。它封装了JDBC。它维护一个“可重复读”的一级缓存。

3. Transaction： 事务

4. 使用hibernate5时，resin4不支持@Table，可以将@Entity的name属性换成表名，删除resin的lib目录下可能有冲突的jar包

# 2. 领域模型（domain model）

## 2.1 映射类型

1. 值类型（value types）：为实体所有，不定义自己的生命周期。

2. 基本类型（basic types）

3. 嵌入类型（Embeddable types）

4. 集合类型（Collection types）

5. 实体类型（Entity types）：是领域类型，和数据库表的列对应。

@Entity(name = "Contact")  
public static class Contact {  
 @Id  
 private Integer id;  
 private Name name;  
 private String notes;  
 private URL website;  
 private boolean starred;  
 //Getters and setters are omitted for brevity  
}  
  
@Embeddable  
public class Name {  
 private String first;  
 private String middle;  
 private String last;  
 // getters and setters omitted  
}

## 2.2 命名策略

hibernate经过两步处理将对象名映射为数据库的表名：通过用户指定（@Column或@Table）或者hibernate自定的策略来生成一个逻辑名称；将逻辑名称映射到定义的表名，策略为：[PhysicalNamingStrategy](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#PhysicalNamingStrategy)

可以通过实现ImplicitNamingStrategy接口或继承ImplicitNamingStrategyJpaCompliantImpl来修改名字对应的策略

## 2.3 基本类型（Basic Types）

基本的值类型一般将数据库中的列映射为一个单一的、非聚合的java类型，可以映射为Java中的装包类型：Integer、BigInteger等

1. @Basic注释：基本类型默认是被javax.persitence.Basic注释的；@Basic定义了两个属性
   * optional(defaults to true):标明该元素是否可以为空；
   * fetch(EAGER,LAZY)：hibernate会忽略该属性。标明是否需要懒加载
2. @Column(name=””): 如果列名和Java中的属性名不一致，可以明确设置列名。其定义的属性名可以参考文档或IDEA的提示
3. BasicTypeRegistry 维护了一个从Java类型到SQL类型的映射表，也可以自己定义BasicType。
4. @Type : 可以修改属性的值映射的类型

### 2.3.1 自定义基本类型

如果想将java.util.BigInteger保存成VARCHAR的列，或者支持全新的类型。有两种方法可以实现：

* 实现一个BasicType并且注册它，因为BasicType接口有许多方法，如果值只是存在一个列中，可以继承AbstractStandarBasicType或者AbstractSingleColumnStandardBasicType.
* 实现UserType，不用注册

两者的代码可参考HibernateDemo com.sogou.umis包

## 2.3.2 枚举类型映射

@Enumerated(EnumType.ORDINAL| EnumType.STRING)可以将枚举类型映射到SQL中。还可以自定义转换函数，将值经过转换后，存入数据库。

public enum PhoneType {  
 *LAND\_LINE*,  
 *MOBILE*;  
}

* EnumType.ORDINAL:将值映射为Integer类型，按顺序排列，对null值，为NULL。对于上例：[INSERT INTO Phone (phone\_number, phone\_type, id) VALUES ('123-456-78990', 2, 1)]
* EnumType.STRING： 将枚举类型的字符串存入数据库，对于上例： [INSERT INTO Phone (phone\_number, phone\_type, id) VALUES ('123-456-78990', 'MOBILE', 1)]

### 2.3.4 映射时间类型

DATE： 代表存储了年月日的时间

TIME：存储了时分秒的时间

TIMESTAMP: 存储DATE和TIME。

[使用@Temporal可以将java.util.Date或者java.util.Calendar映射为DATE](mailto:使用@Temporal可以将java.util.Date或者java.util.Calendar映射为DATE)、TIME、TIMESTAMP

@Column(name = "`timestamp`")  
@Temporal(TemporalType.DATE)  
private Date timestamp;

Java8在java.time包中新增了Date、Time的API

从SQL的Date/Time类型到Java8新增类型的映射为：

DATE ： java.time.LocalDate

TIME : java.time.LocalTime,java.time.OffsetTime

TIMESTAMP: java.time.Instant, java.time.LocalDateTime, java.time.OffsetDateTime, java.time.ZonedDateTime

因为不在是一一对应，所以使用@Temporal将会导致错误。需要指定一个时区，可以在SessionFactory中设定：

settings.put(  
 AvailableSettings.JDBC\_TIME\_ZONE,  
 TimeZone.getTimeZone( "UTC" )  
);

### 2.3.5 属性转换器(AttributeConverter)

对于可以修改的类型，可以使用属性转换器。

public static class MoneyConverter  
 implements AttributeConverter<Money, Long> {  
 @Override  
 public Long convertToDatabaseColumn(Money attribute) {  
 return attribute == null ? null : attribute.getCents();  
 }  
  
 @Override  
 public Money convertToEntityAttribute(Long dbData) {  
 return dbData == null ? null : new Money( dbData );  
 }  
}  
  
@Entity(name = "Account")  
public static class Account {  
 @Id  
 private Long id;  
 private String owner;  
 @Convert(converter = MoneyConverter.class)  
 private Money balance;  
 //Getters and setters are omitted for brevity  
}

一个可以修改的Object运行你修改它内部的结构，Hibernate会将这种修改保存到数据库。

尽管AttributeConverter类型可以被Hibernate脏检查，并被正确处理，但是不可改变的类型的效率更好，应该尽量使用不可修改的类型。

### 2.3.6 列名是SQL关键字

当使用的列名是SQL中的关键字时，可以使用[`或者\”]来转义，Hibernate会根据SQL Dialect来生成合适的SQL语句

@Column(name = "`name`")  
private String name;  
@Column(name = "\"name\"")  
private String name;

【生成的SQL语句】INSERT INTO Product ("name",id) VALUES ('Mobile phone', 1)

还可以配置属性使得全局引用

<property  
 name="hibernate.globally\_quoted\_identifiers"  
 value="true"  
/>

### 2.3.7 @Generated

Hibernate可以在对象实体在更新或者插入后，自动从数据库中获取被@Generated标注的属性。

GenerationTime.ALWAYS：该属性在插入和更新的时候都要生成

GenerationTime.NEVER： 从来不生成

GenerationTime.INSERT: 只在插入的时候生成

@Generated( value = GenerationTime.ALWAYS )  
@Column(columnDefinition =  
 "AS CONCAT(" +  
 " COALESCE(firstName, ''), " +  
 " COALESCE(' ' + middleName1, ''), " +  
 " COALESCE(' ' + middleName2, ''), " +  
 " COALESCE(' ' + middleName3, ''), " +  
 " COALESCE(' ' + middleName4, ''), " +  
 " COALESCE(' ' + middleName5, ''), " +  
 " COALESCE(' ' + lastName, '') " +  
 ")")  
private String fullName;

### 2.3.8 @CreationTimestamp: JVM当前的时间戳

@CreationTimestamp 标记的属性，hibernate在向数据库存储时，会将JVM当前的时间戳的值放入其中。

### 2.3.9 @ValueGenerationType

可以使用@ValueGenerationType来声明生成的属性或者自定义生成器

### 2.3.10 列转换：读写表达式

Hibernate允许你自定义@Basic类型的读写列值的SQL语句。如果数据库提供了加解密的函数，可以像下列一样调用它。

如果使用write表达式，必须用”?”代替其值

@Column(name = "pswd")  
@ColumnTransformer(  
 read = "decrypt( 'AES', '00', pswd )",  
 write = "encrypt('AES', '00', ?)"  
)  
private String password;

如果使用@Columns(cloumns={})则需要使用@Type(type=””)做类型转换，其中type是实现CompositeUserType接口的类。

@ColumnTransformer同@Columns()一起使用时，必须指定forColumn

@Type(type = "org.hibernate.userguide.mapping.basic.MonetaryAmountUserType")  
@Columns(columns = {  
 @Column(name = "money"),  
 @Column(name = "currency")  
})  
@ColumnTransformer(  
 forColumn = "money",  
 read = "money / 100",  
 write = "? \* 100"  
)  
private MonetaryAmount wallet;

### 2.3.11 @Formula

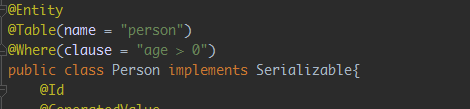
在数据库中计算内容时，可以使用SQL fragment，这种属性是只读的，它的值是由formula计算出来的。SQL fragment可以是任意复杂的，甚至可以使用子查询

@Formula使用的是本地的SQL，可能会有兼容性问题。

@Formula(value = "credit \* rate")  
private Double interest;

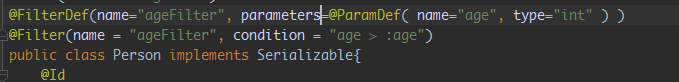
### 2.3.12 @Where

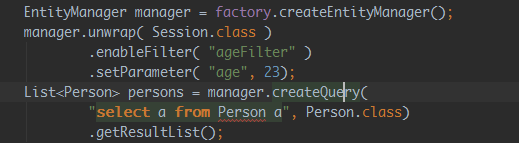
如果要过滤集合，可以使用@Where注释，该注释可以被用于集合和类上



### 2.3.13 @Filter

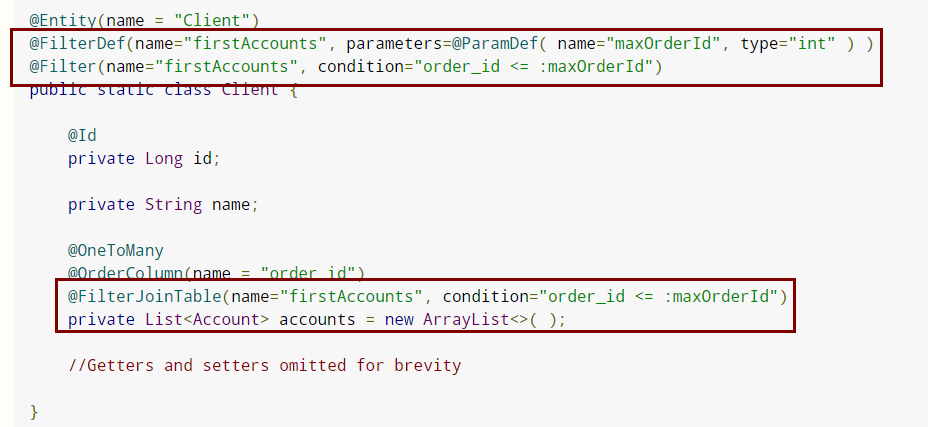
同@Where一样可以过滤集合或者列，不过@Filter允许在运行时指定过滤条件。@Filter在过滤基本类型时，需要将其放置到类上，如果是过滤集合属性，需要将其放到集合定义上

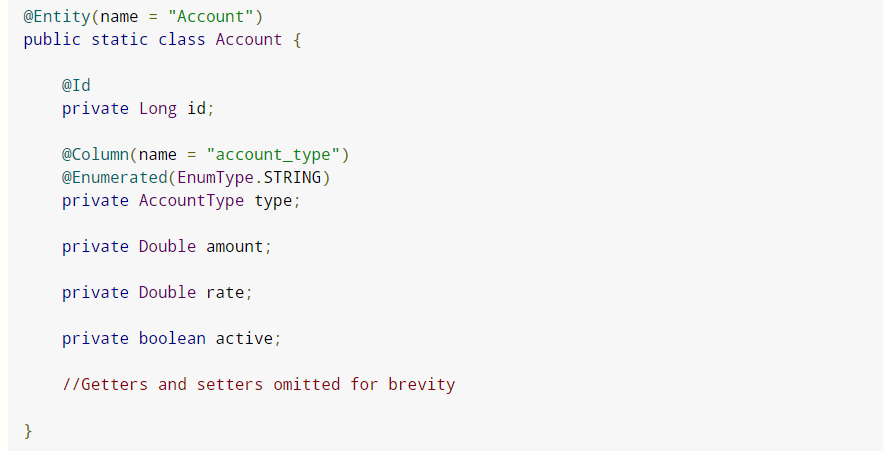




### 2.3.14 @FilterJoinTable

@FilterJoinTable可以被在子表过滤上，使用在@OneToMany集合上





## 2.4 Embeddable类型

在一个Entity中使用了多个相同的Embeddable类型时，需要标明其对应的列。需要使用@AttributeOverride来解决这个问题。



也可以通过修改hibernate的命名策略来解决该问题



## 2.4 Entity类型

hibernate中使用Entity类型必须符合以下条件：

1. Hibernate需要Entity必须有一个无参构造器，可见性最低是package

2. 为保存到数据库的属性声明getters和setters

3. @Entity只有一个name属性，用来指定JPQL queries中的实体名称

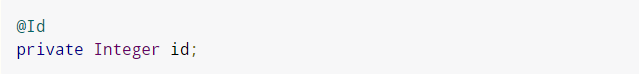
4. @Transient注解的属性，不会被保存到数据库

Hibernate一个核心的特性就是可以通过运行时代理懒加载数据，该特性要求Entity类不能是final或者是实现声明所有getters/setter的接口的类。虽然在hibernate中可以这么做，但是会影响到性能的优化，应该避免将类或者存储的类的属性声明为final。

为持久型属性声明setters和getters，虽然hibernate并不要求，但是最好遵循JavaBean的约定。

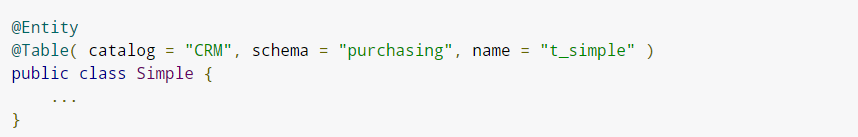
### 2.4.1 identifier

识别符属性不用与数据库定义的主键相对应，但是需要映射到可以唯一标明一列的属性上。Hibernate官方建议将Identifier属性标记在一个可以为空的类型上（非原始类型non-primitive）.



@Entity注解只定义了一个在JPQL查询时用到的实体名的name属性。默认情况下类名就是其实体名。

默认情况下，一个实体映射数据库中的一张表，一个identifier唯一地标识一列，数据库表的名同实体的名一致，也可以使用javax.persitence.Table注解指定表名。不过该方式在resin中不支持。



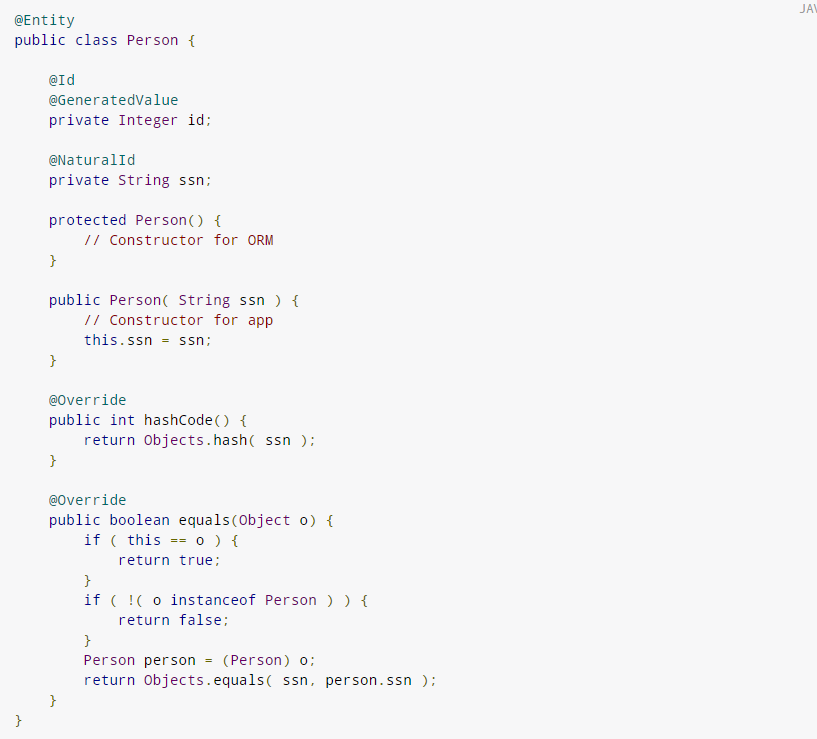
### 2.4.2 equals() 和 hashCode()

1. 当一个类被用作identifier时，必须实现基于id值的equals/hashCode
2. 使用同一个Session多次获取某个Person时，将会返回同一个实例。如果将获取多次的Person都添加到同一个java.util.Set时，将会只添加一个。‘’
3. 使用不同的Session获取Person时，返回的实例是不同的，加到Set中时，取决于是否实现了基于id的equals/hashCode；即使实现了基于id的equals/hashCode也会在 使用generated identifier的类中存在冲突。



上例中，在commit之后p1的id已经被修改。

1. 最终的解决方案是使用一个natural-id或business-key作为equals/hashCode的实现



尽管使用natural-id是equals/hashCode最好的实现，但是或许需要一个唯一约束。

使用identifier来做相等检测时，需要为hashCode提供一个在Entity flushed前后不会变化的量；只用非瞬变的量来比较实体之间是否相等。

### 2.4.3 使用@Version来映射乐观锁

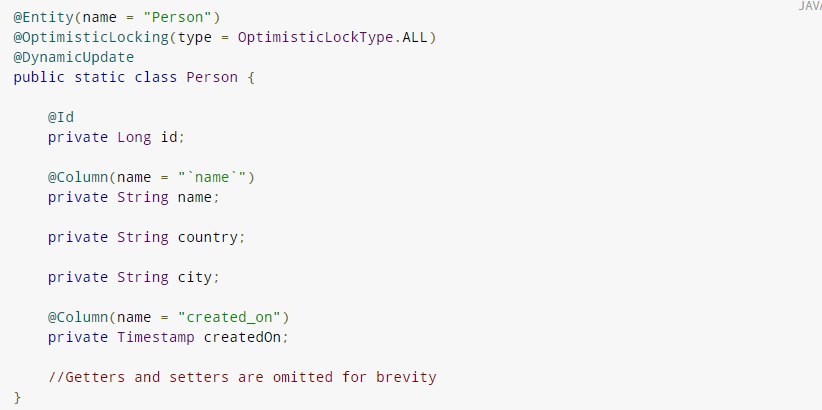
JPA定义了基于version或者timestamp的策略来支持乐观锁。只需要使用javax.persitence.Version注解到持久属性上，就能启用乐观锁。有效的类型有：int、Integer、short、Short、long、Long、java.sql.Timestamp

在许多场合使用@Version属性是有效的，但是你可以依赖实际的列的值来防止丢失更新。

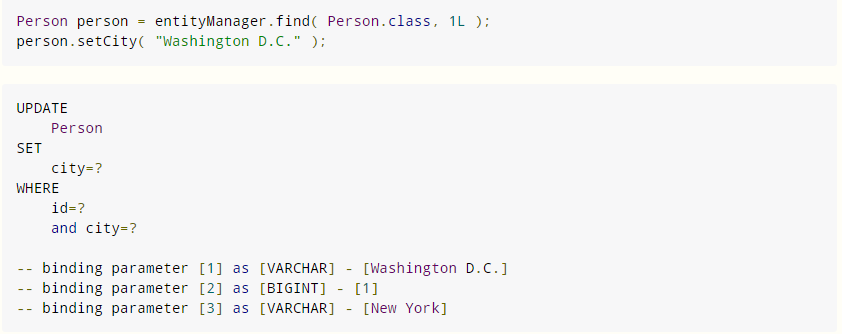
Hibernate支持使用所有的实体属性或者被改变的实体属性作为查询语句，防止丢失更新。使用@OptimisticLocking(type = OptimisticLockType.ALL)注解来使用这种方式。使用OptimisticLockType.ALL时，必须使用@DynamicUpdate

OptimisticLockTypes有四个可以用的值：

* NONE: 乐观锁被禁用，有@Version也不起作用
* VERSION: 基于@Version来启用乐观锁
* ALL：在UPDATE/DELETE时，where条件使用所有的fields
* DIRTY: 在UPDATE/DELETE时，where条件使用dirty filelds，即需要改变的field在where条件中



使用OptimisticLockType.DIRTY时，也要使用@DynamicUpdate和@SelectBeforeUpdate。在更新的时候，会把更新的内容放置到where条件中。



### 2.4.5 访问策略

属性名以两个大写字符开头时，在HQL中会有奇怪的功能性问题。

使用@Transient注解将标记的属性从持久化中移除

1. 基于字段的注解 (field-based): 将注解放置到实体类的字段前面（建议使用该注解），使用注入的访问策略，而不是使用get方式访问
2. 基于属性的注解 (property-based): 将注解放置在实体类相应的getter方法上，使用get和set的访问策略

## 2.5 Identifier（标识符）

标识符用来唯一标记集合中的某一实体。

* UNIQUE：值必须唯一标识每行
* NOT NULL： 值不能为null，对于符合类型的id，每部分都不能为null
* IMMUTABLE： 插入之后就不能修改。这条是一个普遍的指导，不是硬性规定。

### 2.5.1 简单的标识符

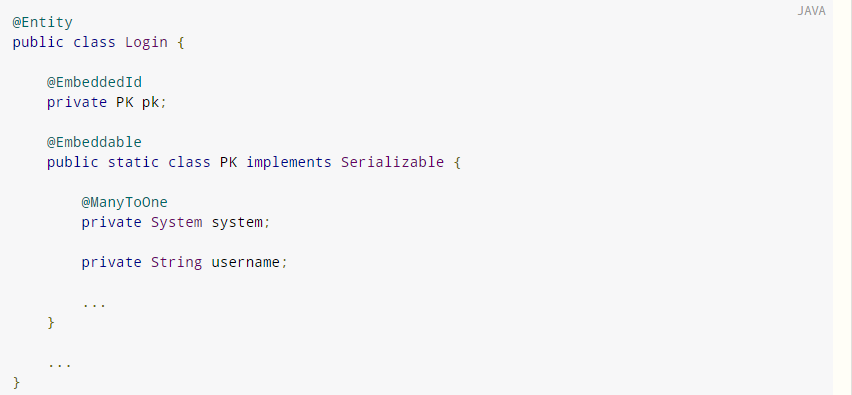
简单标记符(Simple Identifiers)可以使用java.persistence.Id注解来标记，映射为一个基本类型。

只有一下类型可以用来做简单标记符：

Java原始数据类型；原始类型的装箱类型；String；java.util.Date; java.sql.Date; java.math.BigDecimal; java.math.BigInteger

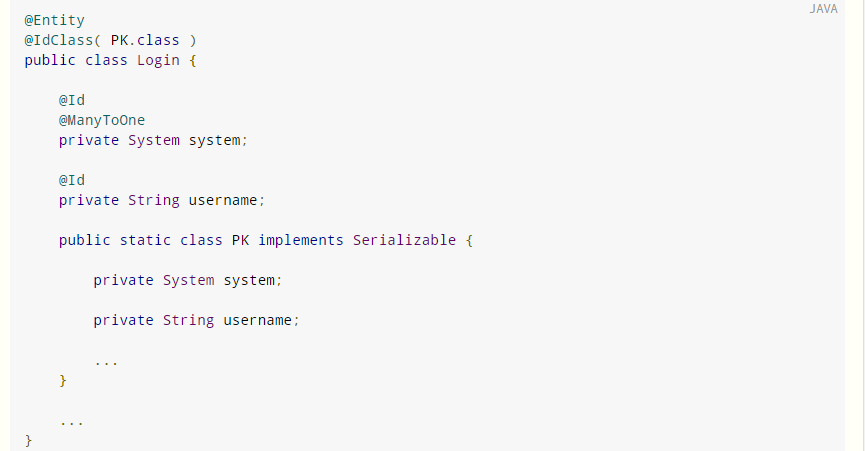
### 2.5.2 复合标记符 @EmbeddedId

使用EmbeddedId来注解复合标记符只是简单定义了一个多值的嵌套类来生成一个标记符。EmbeddedId可以包含ManyToOne的属性，在Entity中直接使用这些属性



### 2.5.3 复合标记符 @IdClass

与EmbeddedId的不同在与 IdClass使用不侵入式的做法



### 2.5.4 标识符生成

hibernate可以生成很多类型的标识符.

使用javax.persistence.GeneratedValue注解时，最重要的是指定表名值是如何生成的javax.persistence.GeneratedType属性。

其中javax.persistence.GeneratedType的取值有：

* AUTO: hibernate自己选择一个合适的生成策略

先去查找标识符属性的java类型，如果类型是UUID，hibernate会使用一个UUID标记符；如果是数字类型（Long，Integer），hibernate将会使用IdGeneratorStrategyInterperter来生成一个标识符，该策略有两个实现：

FallbackInterpreter在5.0之后，是默认的生成策略。当使用该策略时，AUTO将会使用SequenceStyleGenerator，如果数据库支持序列，将会使用SEQUENCE生成器；否则，将会使用TABLE生成器（创建一个新表）；

LegacyFallbackInterpreter将会使用数据库默认的生成策略（identity或者sequence），5.0后将hibernate.id.new\_generator\_mappings属性配置为false，启用该策略

* IDENTITY: 将会使用数据库标识符生成主键的值

使用该策略，在插入之前是不知道其唯一标识符的，会破坏上下文的持久性。hibernate建议使用其他的生成策略。

使用该策略会导致不能使用JDBC的批处理策略。

* SEQUENCE: 使用数据库序列来生成主键
* TABLE: 使用数据库表生成主键

通过生成一个hibernate\_sequences(可以自己指定)来生成id

@Id  
@GeneratedValue(generator = "uuid")  
**@GenericGenerator(  
 name = "uuid",  
 strategy = "org.hibernate.id.UUIDGenerator",  
 parameters = {  
 @Parameter(  
 name = "uid\_gen\_strategy\_class",  
 value = "org.hibernate.id.uuid.CustomVersionOneStrategy"  
 )  
 }  
)**private UUID id;

### 2.5.5 优化器

使用优化器可以减少从数据库获取生成标识符所需要信息的次数。

* none: 不使用优化器，每次需要标识符值的时候，hibernate都要从数据库中获取
* pooled-lo: 要求数据库序列/表每次自增的值都大于1. 如果自增为20，那么每次获取后，可以使用20次
* pooled: 和pooled类似，只是使用的池子中的大值
* hilo; legacy-hilo: 自定义生成标识符池子的算法，不推荐使用

优化器需要结合@GenericGenerator使用

@Id  
@GeneratedValue(  
 strategy = GenerationType.*SEQUENCE*,  
 generator = "product\_generator"  
)  
**@GenericGenerator(  
 name = "product\_generator",  
 strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",  
 parameters = {  
 @Parameter(name = "sequence\_name", value = "product\_sequence"),  
 @Parameter(name = "initial\_value", value = "1"),  
 @Parameter(name = "increment)size", value = "3"),  
 @Parameter(name = "optimizer", value = "pooled-lo")  
 }  
)**private Long id1;

### 2.5.6 驱动标识符

实体可以从many-to-one或者one-to-one借来标识符，使用的注解为@MapsId/@PrimaryKeyJoinColumn

## 2.6 关联关系

关联关系描述了基于数据库语义的实体之间的关系。

### 2.6.1 @ManyToOne

@ManyToOne是一种比较普遍的关系，映射了数据库中的关系，建立了子实体和父实体之间的联系。

### 2.6.2 @OneToMany

@OneToMany的关系连接着一个父实体和多个子实体。如果@OneToMany没有一个@ManyToOne对应的话，那么@OneToMany是单向的，否则是双向的。

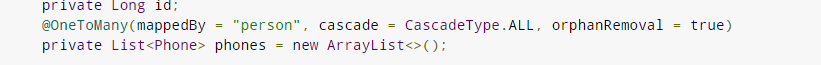
当使用单向的@OneToMany时，Hibernate将会使用一个中间table来关联两个实体。

使用级联(@OneToMany(cascade=CascadeType.ALL)时，保存父实体时，会默认将相应的子实体也保存。当从子实体集合中移除一个子实体时，link table相关的列也会被删除。

当移除子实体时，使用单向的相关连接效率比较低。Hibernate会删除所有的子实体，然后保存那些可以在内存上下文中存在的值。

在这种情况下，使用双向的@OneToMany会更有效

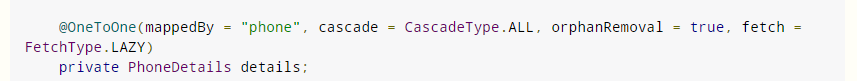
双向的@OneToMany, @OneToMany需要在子实体中有@ManyToOne与之相对应。每个双向关联必须有一个所有边（子实体），另一个引用这个 (mappedBy)

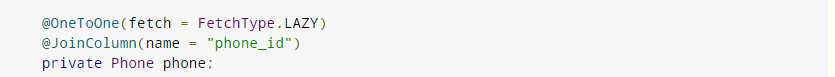




### 2.6.3 @OneToOne

@OneToOne可以是单向的，也可以是双向的。单向的与数据库中的外键情形相似，client-side存储关系。双向关联在父实体的@OneToOne中使用mappedBy





### 2.6.4 @ManyToMany

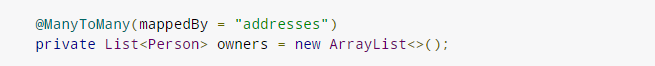
@ManyToMany关联关系，需要一个中间的表来表示两个实体间的管理关系。

单向的@ManyToMany，就像@OneToMany关系似的，所有者控制着关联表。 当从@ManyToMany集合中移除实体时，Hibernate从关联表中删除相应的实体。删除实体时，会先删除关系表中所有实体对应的关系，再将原来保存的关系存储区来。不能将CascadeType.REMOVE加到多对多关系上，因为一个实体可能对应着多个实体。



双向的@ManyToMany有一个所有侧和一个mappedBy侧。为了保持两侧的同步，最好提供添加和删除子实体的方法





### 2.6.5 @NotFound(action = NotFoundAction.IGNORE)

当子实体中所指向的父实体不存在时，使用@NotFound，hibernate会忽略异常并将父实体的引用赋值为null

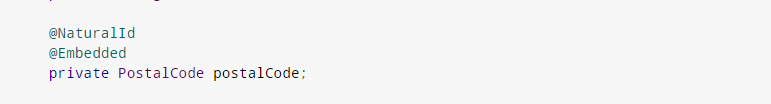
## 2.7 Collections

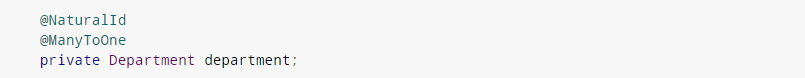
如果在Hibernate中使用集合类型，最好使用集合类型的接口，因为Hibernate返回的类型可能是Hibernate实现了集合接口的集合类型

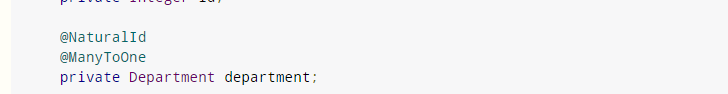
......

## 2.8 Natural Ids 自然的唯一标识

Natural ids是指本身具有唯一性的属性。Hibernate提供了高效的API来根据Natural id查询实体。

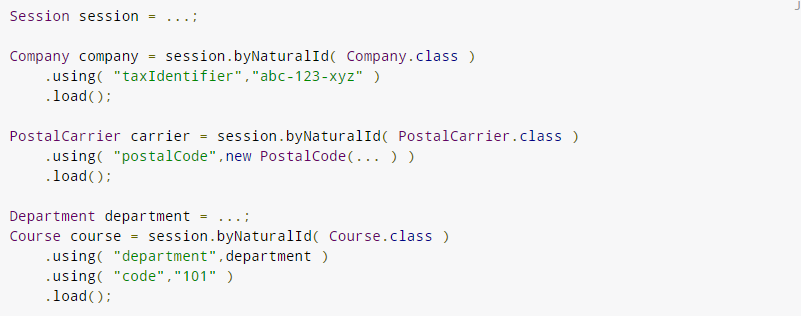






### 2.8.1 Natural Id API

Hibernate为natural id提供了专门的API，session.byNaturalId。 如果实体中没有定义natural id，使用该方法将会抛出异常。



* load(): 获取一个被初始化的实体引用
* getReference(): 获取一个可能没有被初始化的实体引用。如果该实体已经在Session中获取到了，直接返回该实体的引用。若实体没有被当前Session加载，如果实体支持代理，会返回未初始化的代理，否则会从数据库中加载。

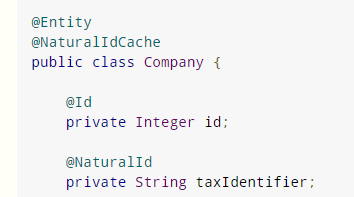
如果实体中只有一个@NaturalId标记的属性，可以使用Session.bySimpleNaturalId().load()来获取。

### 2.8.2 Natural Id – 可修改和缓存

当一个Natural Id可被修改时，需要使用 @NaturalId(mutalbe=true).

Hibernate在内部维护着一个从natural id到实体identifier的映射，如果natural id的值改变了，该映射直到flush之前，数据都是过期的。为了解决这个问题，Hibernate会试着在load或者getReference时，调整内部的映射。这些调整会带来性能上的缺陷。如果一个应用可以确定与Session关联的可变的natural id都没有发生变化，可以使用 setSynchronizationEnabled(false)来关闭可变的natural id的检测。

我们可以将NaturalId-to-PK的映射缓存到二级缓存中，如果二级缓存打开的话。



## 2.9 动态映射模型

保存的实体类型没必要一定是POJO/JavaBean的类型。Hibernate指定动态模型（Map），使用这种方式只需要写映射文件。



# 3. Bootstrapping

bootstrapping 一般指初始化和开始一个组件。在Hibernate中指创建一个具有完整功能的SessionFactory或者对JPA的EntityManagerFactory实体。

## 3.1 JPA Bootstrapping

用Bootstrapping hibernate作为JPA的提供者，可以使用JPA特定的方式。标准的方法虽然在特定的环境下有些限制，但是除了在特定环境下，还是非常土建使用JPA标准的bootstrapping。

### 3.1.1 JPA方式的Bootstrapping

在JPA中，最终启动的是一个javax.persistence.EntityManagerFactory实例。JPA根据从EntityManagerFactory获取javax.persistence.EntityManager的方式定义了两种主要的标准化启动方式： EE和SE。

EE实现了代表应用管理和注入持久化上下文的容器。SE是其他所有的实现。

## 3.2 Hibernate的Bootstrapping

Hibernate创建SessionFactory的过程。

首先创建一个Hibernate在启动和运行时使用的services的存储器ServiceRegistry。

如果想要更改BootStrapServiceRegistry的创建过程，可以使用org.hibernate.boot.registry.BootstrapServiceRegistry

还可以使用org.hibernate.boot.registry.StandardServiceRegistry，需要通过StandardServiceRegistryBuilder配置StandardServiceRegistry

# 6. Flushing

Flushing 是将持久化上下文同数据库同步的操作。EntityManager和Hibernate的Session提供了一系列的方法，这些方法可以改变实体的存储状态。

持久化上下文使用队列储存搜索实体状态的修改，同write-behind缓存一样。首先将变化存储在内存中，之后在flush期间将变化同步到数据库中。flush操作将每个实体的状态变化转换成相应的INSERT，UPDATE或者DELETE语句。

Hibernate定义了几种刷新的策略，JPA只定义了两种：AUTO和COMMIT，通过session.setFlushMode(FlushModeType.COMMIT)设置。

* ALWAYS：每个query之前都flush session
* AUTO： 默认的策略，只在必要的时候flush session，会在三种情况下触发：
  + 在Transaction提交之前
  + 在执行与实体相关的JPQL/HQL查询之前
  + 在执行任何原生的SQL查询之前
* COMMIT： Session尝试着在当前Transaction提交之前flush，但是也可能会过早地flush
* MANUAL： 应用程序控制着Session的flush，必须明确调用Session.flush()来将变化保存到内存中。

## 6.1 Flush操作顺序

Hibernate执行SQL语句的顺序不依赖于实体操作的顺序，是以ActionQueue中的顺序来执行的。该顺序如下：

1. OrphanRemovalAction (Delete)

2. EntityInsertAction or EntityIdentityInsertAction (Insert)

3. EntityUpdateAction (Update)

4. CollectionRemoveAction

5. CollectionUpdateAction

6. CollectioniRecreateAction

7. EntityDeleteAction (DELETE)

# 8. Transactions 和 并发控制

对于不同的O/R映射，transactions有不同的含义。在大多数情况下，这些定义时一致的，但是总有些不一样：

* 可以引用数据库的事物
* 可以引用与持久化上下文相关的事物逻辑概念
* 可以引用由原型模式定义的工作单元

## 8.1 物理事物

Hibernate使用JDBC API来处理持久化。JDBC的事物中有两种良好的机制：JDBC和JTA。

Hibernate直接使用JDBC连接和JTA资源，不添加任何额外的锁。Hibernate也不锁定内存中的对象。使用Hibernate时，在数据库事务隔离层定义的行为不会改变。Hibernate的Session像是对加载的实体提供重复读的事务范围的缓存。

为了减少数据库中的锁竞争，数据库事务要尽可能地短。长数据库事务将会减低数据库的并发。在用户级时，不打开数据库事务，等用户级结束的时候打开。