Hibernate中domain配置文件

将主键改成自增长类型：

Oracle设置<generator class="**increment**" />

*SQLServer 2000*设置<generator class="**identity**" />

―――――――――例如*SQLServer 2000中*――――――――――――――――――

<id name="userid" type="java.lang.Integer">

            <column name="userid" />

            <generator class="identity" />

 </id>

―――――――――――――――――Oracle*中*――――――――――――――――――

<id name="userid" type="java.lang.Integer">

            <column name="userid" />

            <generator class="increment" />

 </id>

hibernate提供了产生自动增长类型主键的多种策略,这里以increment为例说明具体用法:   
1、在数据库中建立表，主键名称为ID,类型为varchar2（字符串型）   
2、在\*\*.hbm.xml（hibernate映射文件）中配置如下   
<class name="com.jat.bisarea.ho.Test" table="BA\_VVTEST">   
<id name="id" type="int" column="ID">   
//该句指定使用hibernate自带的increment策略生成主键   
<generator class="increment"/>   
</id>   
<property name="uname" type="java.lang.String" column="UNAME"/>   
</class>   
3、在java文件中对表增加记录时，只需添加除ID外的其他字段，然后save即可，相关java代码如下：   
Session s = HibernateUtil.currentSession();   
Transaction tx = s.beginTransaction();   
Test test = new Test();   
String uname = httpServletRequest.getParameter("uname");   
test.setUname(uname);   
//只需对uname进行set，id由hibernate生成   
s.save(test);   
tx.commit();   
4、使用其它策略的方法基本一致，例如hilo、seqhilo等   
Generator 为每个 POJO 的实例提供唯一标识。一般情况，我们使用“native”。class 表示采用由生成器接口net.sf.hibernate.id.IdentifierGenerator 实现的某个实例，其中包括：   
**“assigned”**  
主键由外部程序负责生成，在 save() 之前指定一个。   
**“hilo”**  
通过hi/lo 算法实现的主键生成机制，需要额外的数据库表或字段提供高位值来源。   
**“seqhilo”**   
与hilo 类似，通过hi/lo 算法实现的主键生成机制，需要数据库中的 Sequence，适用于支持 Sequence 的数据库，如Oracle。   
**“increment”**  
主键按数值顺序递增。此方式的实现机制为在当前应用实例中维持一个变量，以保存着当前的最大值，之后每次需要生成主键的时候将此值加1作为主键。这种方式可能产生的问题是：不能在集群下使用。   
**“identity”**  
采用数据库提供的主键生成机制。如DB2、SQL Server、MySQL 中的主键生成机制。   
**“sequence”**  
采用数据库提供的 sequence 机制生成主键。如 Oralce 中的Sequence。   
**“native”**  
由 Hibernate 根据使用的数据库自行判断采用 identity、hilo、sequence 其中一种作为主键生成方式。   
**“uuid.hex”**  
由 Hibernate 基于128 位 UUID 算法生成16 进制数值（编码后以长度32 的字符串表示）作为主键。   
**“uuid.string”**  
与uuid.hex 类似，只是生成的主键未进行编码（长度16），不能应用在 PostgreSQL 数据库中。   
**“foreign”**  
使用另外一个相关联的对象的标识符作为主键。

1. native   
我最常用的。可以保证多个数据库之间的可移植性。但是有可能有时候会有问题：因为不能控制id值，在数据倒表的时候可能无法满足业务需要。  
2. sequence  
这种地方可以解决上面用native时候的问题，但是需要堆数据库做一些其他配置。  
3. uuid  
理论上可以保证多个数据库生成的ID在一个系统里唯一，有时候挺有用。但是效率稍微低点（其实都无所谓）。  
4. increment  
最好不要用。如果有其他程序访问、修改数据库，那就恐怖了。  
5. assigned  
没怎么用过。一般不会用手工方式赋值主键，除非有特殊的需求。  
6.  foreign  
在one-to-one的时候可能会用到。  
7. 在使用数据库自动生成主键的时候，SQL语句会有所不同：有些数据库不许你填主键，有些要求你该字段必须为null，有些会完全忽略你写的主键的值。

订单号如何生成

方法1

访问量小可行

Date d=new Date();  
System.out.println(d.getTime());//这里就得到了唯一的编号。

方法2

首先，订单号有3个性质：1.唯一性 2.不可推测性 3.效率性  
  
唯一性和不可推测性不用说了，效率性是指不能频繁的去数据库查询以避免重复。  
况且满足这些条件的同时订单号还要足够的短。  
我在java下定制的订单号生成方式如下：  
int r1=(int)([Math.random](https://www.baidu.com/s?wd=Math.random&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4P17WrARvnj61nWRvmh7B0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHfYPHbknjm3" \t "_blank)()\*(10));//产生2个0-9的随机数  
int r2=(int)([Math.random](https://www.baidu.com/s?wd=Math.random&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y4P17WrARvnj61nWRvmh7B0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHfYPHbknjm3" \t "_blank)()\*(10));  
long now = System.currentTimeMillis();//一个13位的时间戳  
String paymentID =String.valueOf(r1)+String.valueOf(r2)+String.valueOf(now);// 订单ID  
//String 类别中已经提供了将基本数据型态转换成 String 的 static 方法 也就是 String.valueOf() 这个参数多载的方法  
目前规则来看，两个人在同一微秒提交订单重复的概率为1%

方法3

类用于产生32位的绝对全球唯一的编号，类似于hibernate中uuid生成方式

package com.anxin.utils;

import java.io.Serializable;

import java.net.InetAddress;

/\*\*

\* 生成类似hibernate中uuid 32位主键序列

\*

\* @version: V1.0

\*/

public class UUIDGenerator {

private static final int IP;

public static int IptoInt(byte[] bytes) {

int result = 0;

for (int i = 0; i < 4; i++) {

result = (result << 8) - Byte.MIN\_VALUE + (int) bytes[i];

}

return result;

}

static {

int ipadd;

try {

ipadd = IptoInt(InetAddress.getLocalHost().getAddress());

} catch (Exception e) {

ipadd = 0;

}

IP = ipadd;

}

private static short counter = (short) 0;

private static final int JVM = (int) (System.currentTimeMillis() >>> 8);

public UUIDGenerator() {

}

public static int getJVM() {

return JVM;

}

public static short getCount() {

synchronized (UUIDGenerator.class) {

if (counter < 0)

counter = 0;

return counter++;

}

}

public static int getIP() {

return IP;

}

public static short getHiTime() {

return (short) (System.currentTimeMillis() >>> 32);

}

public static int getLoTime() {

return (int) System.currentTimeMillis();

}

private final static String sep = "";

public static String format(int intval) {

String formatted = Integer.toHexString(intval);

StringBuffer buf = new StringBuffer("00000000");

buf.replace(8 - formatted.length(), 8, formatted);

return buf.toString();

}

public static String format(short shortval) {

String formatted = Integer.toHexString(shortval);

StringBuffer buf = new StringBuffer("0000");

buf.replace(4 - formatted.length(), 4, formatted);

return buf.toString();

}

public static String generate() {

return String.valueOf(new StringBuffer(36).append(format(getIP())).append(sep)

.append(format(getJVM())).append(sep)

.append(format(getHiTime())).append(sep)

.append(format(getLoTime())).append(sep)

.append(format(getCount())).toString());

}

public static void main(String args[]){

System.out.println(UUIDGenerator.generate());

}

}

Hibernate中的延迟加载

延迟加载机制是为了避免一些无谓的性能开销而提出来的，所谓延迟加载就是当在真正需要数据的时候，才真正执行数据加载操作。在Hibernate中提供了对实体对象的延迟加载以及对集合的延迟加载，另外在Hibernate3中还提供了对属性的延迟加载。下面我们就分别介绍这些种类的延迟加载的细节。   
A、实体对象的延迟加载：   
如果想对实体对象使用延迟加载，必须要在实体的映射配置文件中进行相应的配置，如下所示：   
<hibernate-mapping>   
<class name=”net.ftng.entity.user” table=”user” lazy=”true”>   
……   
</class>   
</hibernate-mapping>   
通过将class的lazy属性设置为true，来开启实体的延迟加载特性。如果我们运行下面的代码：   
User user=(User)session.load(User.class,”1”);（1）   
System.out.println(user.getName());（2）   
当运行到(1)处时，Hibernate并没有发起对数据的查询，如果我们此时通过一些调试工具(比如JBuilder2005的Debug工具)，观察此时user对象的内存快照，我们会惊奇的发现，此时返回的可能是User$EnhancerByCGLIB$$bede8986类型的对象，而且其属性为null,这是怎么回事？还记得前面我曾讲过session.load()方法，会返回实体对象的代理类对象，这里所返回的对象类型就是User对象的代理类对象。在Hibernate中通过使用CGLIB,来实现动态构造一个目标对象的代理类对象，并且在代理类对象中包含目标对象的所有属性和方法，而且所有属性均被赋值为null。通过调试器显示的内存快照，我们可以看出此时真正的User对象，是包含在代理对象的CGLIB$CALBACK\_0.target属性中，当代码运行到（2）处时，此时调用user.getName()方法，这时通过CGLIB赋予的回调机制，实际上调用CGLIB$CALBACK\_0.getName()方法，当调用该方法时，Hibernate会首先检查CGLIB$CALBACK\_0.target属性是否为null，如果不为空，则调用目标对象的getName方法，如果为空，则会发起数据库查询，生成类似这样的[SQL语句](https://www.baidu.com/s?wd=SQL%E8%AF%AD%E5%8F%A5&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBujm4uHI9nAnvuW63myP90ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbLnjmkPWf1" \t "_blank)：select \* from user where id=’1’;来查询数据，并构造目标对象，并且将它赋值到CGLIB$CALBACK\_0.target属性中。   
这样，通过一个中间代理对象，Hibernate实现了实体的延迟加载，只有当用户真正发起获得实体对象属性的动作时，才真正会发起数据库查询操作。所以实体的延迟加载是用通过中间代理类完成的，所以只有session.load()方法才会利用实体延迟加载，因为只有session.load()方法才会返回实体类的代理类对象。   
B、 集合类型的延迟加载：   
在Hibernate的延迟加载机制中，针对集合类型的应用，意义是最为重大的，因为这有可能使性能得到大幅度的提高，为此Hibernate进行了大量的努力，其中包括对JDK Collection的独立实现，我们在一对多关联中，定义的用来容纳关联对象的Set集合，并不是java.util.Set类型或其子类型，而是net.sf.hibernate.collection.Set类型，通过使用自定义集合类的实现，Hibernate实现了集合类型的延迟加载。为了对集合类型使用延迟加载，我们必须如下配置我们的实体类的关于关联的部分：   
  
<hibernate-mapping>   
<class name=”net.ftng.entity.User” table=”user”>   
…..   
<set name=”addresses” table=”address” lazy=”true” inverse=”true”>   
<key column=”user\_id”/>   
<one-to-many class=”net.ftng.entity.Arrderss”/>   
</set>   
</class>   
</hibernate-mapping>   
通过将<set>元素的lazy属性设置为true来开启集合类型的延迟加载特性。我们看下面的代码：   
User user=(User)session.load(User.class,”1”);   
Collection addset=user.getAddresses(); (1)   
Iterator it=addset.iterator(); (2)   
while(it.hasNext()){   
Address address=(Address)it.next();   
System.out.println(address.getAddress());   
}   
当程序执行到(1)处时，这时并不会发起对关联数据的查询来加载关联数据，只有运行到(2)处时，真正的数据读取操作才会开始，这时Hibernate会根据缓存中符合条件的数据索引，来查找符合条件的实体对象。   
这里我们引入了一个全新的概念——数据索引，下面我们首先将接一下什么是数据索引。在Hibernate中对集合类型进行缓存时，是分两部分进行缓存的，首先缓存集合中所有实体的id列表，然后缓存实体对象，这些实体对象的id列表，就是所谓的数据索引。当查找数据索引时，如果没有找到对应的数据索引，这时就会一条select SQL的执行，获得符合条件的数据，并构造实体对象集合和数据索引，然后返回实体对象的集合，并且将实体对象和数据索引纳入Hibernate的缓存之中。另一方面，如果找到对应的数据索引，则从数据索引中取出id列表，然后根据id在缓存中查找对应的实体，如果找到就从缓存中返回，如果没有找到，在发起select SQL查询。在这里我们看出了另外一个问题，这个问题可能会对性能产生影响，这就是集合类型的缓存策略。如果我们如下配置集合类型：   
<hibernate-mapping>   
<class name=”net.ftng.entity.User” table=”user”>   
…..   
<set name=”addresses” table=”address” lazy=”true” inverse=”true”>   
<cache usage=”read-only”/>  
<key column=”user\_id”/>   
<one-to-many class=”net.ftng.entity.Arrderss”/>   
</set>   
</class>   
</hibernate-mapping>   
这里我们应用了<cache usage=”read-only”/>配置，如果采用这种策略来配置集合类型，Hibernate将只会对数据索引进行缓存，而不会对集合中的实体对象进行缓存。如上配置我们运行下面的代码：   
User user=(User)session.load(User.class,”1”);   
  
Collection addset=user.getAddresses();   
Iterator it=addset.iterator();   
while(it.hasNext()){   
Address address=(Address)it.next();   
System.out.println(address.getAddress());   
}   
System.out.println(“Second query……”);   
User user2=(User)session.load(User.class,”1”);   
Collection it2=user2.getAddresses();   
while(it2.hasNext()){   
Address address2=(Address)it2.next();   
System.out.println(address2.getAddress());   
}   
运行这段代码，会得到类似下面的输出：   
Select \* from user where id=’1’;   
Select \* from address where user\_id=’1’;   
Tianjin   
Dalian   
Second query……   
Select \* from address where id=’1’;   
  
Select \* from address where id=’2’;   
Tianjin   
Dalian   
我们看到，当第二次执行查询时，执行了两条对address表的查询操作，为什么会这样？这是因为当第一次加载实体后，根据集合类型缓存策略的配置，只对集合数据索引进行了缓存，而并没有对集合中的实体对象进行缓存，所以在第二次再次加载实体时，Hibernate找到了对应实体的数据索引，但是根据数据索引，却无法在缓存中找到对应的实体，所以Hibernate根据找到的数据索引发起了两条select SQL的查询操作，这里造成了对性能的浪费，怎样才能避免这种情况呢？我们必须对集合类型中的实体也指定缓存策略，所以我们要如下对集合类型进行配置：   
<hibernate-mapping>   
<class name=”net.ftng.entity.User” table=”user”>   
…..   
<set name=”addresses” table=”address” lazy=”true” inverse=”true”>   
<cache usage=”read-write”/>   
<key column=”user\_id”/>   
<one-to-many class=”net.ftng.entity.Arrderss”/>   
</set>   
</class>   
</hibernate-mapping> 此时Hibernate会对集合类型中的实体也进行缓存，如果根据这个配置再次运行上面的代码，将会得到类似如下的输出：   
Select \* from user where id=’1’;   
Select \* from address where user\_id=’1’;   
Tianjin   
Dalian   
Second query……   
Tianjin   
Dalian   
这时将不会再有根据数据索引进行查询的SQL语句，因为此时可以直接从缓存中获得集合类型中存放的实体对象。  
C、 属性延迟加载：   
在Hibernate3中，引入了一种新的特性——属性的延迟加载，这个机制又为获取高性能查询提供了有力的工具。在前面我们讲大数据对象读取时，在User对象中有一个resume字段，该字段是一个java.sql.Clob类型，包含了用户的简历信息，当我们加载该对象时，我们不得不每一次都要加载这个字段，而不论我们是否真的需要它，而且这种大数据对象的读取本身会带来很大的性能开销。在Hibernate2中，我们只有通过我们前面讲过的面性能的粒度细分，来分解User类，来解决这个问题（请参照那一节的论述），但是在Hibernate3中，我们可以通过属性延迟加载机制，来使我们获得只有当我们真正需要操作这个字段时，才去读取这个字段数据的能力，为此我们必须如下配置我们的实体类：   
<hibernate-mapping>   
<class name=”net.ftng.entity.User” table=”user”>   
……   
<property name=”resume” type=”java.sql.Clob” column=”resume” lazy=”true”/>   
</class>   
</hibernate-mapping>   
通过对<property>元素的lazy属性设置true来开启属性的延迟加载，在Hibernate3中为了实现属性的延迟加载，使用了类增强器来对实体类的[Class文件](https://www.baidu.com/s?wd=Class%E6%96%87%E4%BB%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBujm4uHI9nAnvuW63myP90ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbLnjmkPWf1" \t "_blank)进行强化处理，通过增强器的增强，将CGLIB的回调机制逻辑，加入实体类，这里我们可以看出属性的延迟加载，还是通过CGLIB来实现的。CGLIB是Apache的一个开源工程，这个类库可以操纵java类的字节码，根据字节码来动态构造符合要求的类对象。根据上面的配置我们运行下面的代码：  
String sql=”from User user where user.name=’zx’ ”;   
Query query=session.createQuery(sql); (1)   
List list=query.list();   
for(int i=0;i<list.size();i++){   
User user=(User)list.get(i);   
System.out.println(user.getName());   
System.out.println(user.getResume()); (2)   
}   
当执行到(1)处时，会生成类似如下的SQL语句：   
Select id,age,name from user where name=’zx’;   
这时Hibernate会检索User实体中所有非延迟加载属性对应的字段数据，当执行到(2)处时，会生成类似如下的SQL语句：   
Select resume from user where id=’1’;   
这时会发起对resume字段数据真正的读取操作。