

hibernate杂谈

作者: jinnianshilongnian

<http://jinnianshilongnian.iteye.com>

hibernate杂谈

目 录

1. hibernate杂谈

1.1 Hibernate存取JSON数据（换一种思路来存取数据） 3

1.2 Hibernate 关系映射 总结整理 12

1.3 Hibernate 二级缓存 总结整理 31

1.4 Hibernate自定义类型 对象--->序列化为字符串 存储 48

1.5 Hibernate自定义类型 集合--->字符串 存储 54

1.1 Hibernate存取JSON数据（换一种思路来存取数据）

发表时间: 2012-04-25 关键字: hibernate

一、场景

```
public class OrderModel {  
  
    private List<String> favorableDescList;  
  
}
```

订单中会存储一些优惠信息，方便页面展示时使用，如：

- 1、满100减50
- 2、参与【老会员真情回馈——精品课程体验活动】，仅需支付200.00学币
- 3、【Oracle + PL/SQL 实战】套装课程的【抢购】活动，优惠120.00学币
-等等

如图所示，我们在页面给用户展示他们参与的优惠信息：

您参与的优惠活动

【Oracle + PL/SQL 实战】套装课程的【抢购】活动，优惠120.00学币
【jquery使用基础】套装课程的【抢购】活动，优惠50.00学币

二、分析

如上优惠信息有如下特点：

- 1、只用于展示，不会涉及修改；
- 2、一旦订单支付成功，不会再改变；

3、数据量不会很大。

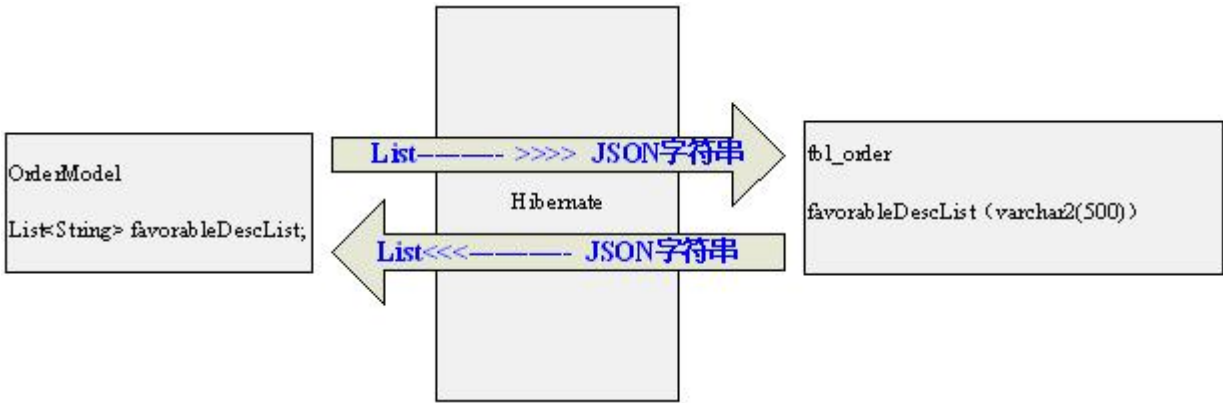
三、解决方案

1、最简单的解决方案是关联表：



但这种解决方案需要连表进行查询，感觉是没有必要的，毕竟只是展示数据，用关联表有点杀鸡用牛刀的感觉。

2、JSON解决方案：



通过如上思路我们可以解决许多类似的问题。

3、代码示例：

1、模型类：

```
public class OrderModel {  
    @Type(type = "cn.javass.framework.hibernate.type.JsonType") //①
```

```
private List<String> favorableDescList;  
}
```

①处使用我们自定义的Hibernate类型来进行转换，上边代码只有一部分

2、自定义JsonType

```
package cn.javass.framework.hibernate.type;  
//省略import  
public class JsonType implements UserType, Serializable {  
    private String json;  
    @Override  
    public int[] sqlTypes() {  
        return new int[] {Hibernate.STRING.sqlType()};  
    }  
    @Override  
    public Class returnedClass() {  
        return JsonList.class;  
    }  
    @Override  
    public boolean equals(Object o, Object o1) throws HibernateException {  
        if (o == o1) {  
            return true;  
        }  
        if (o == null || o1 == null) {  
            return false;  
        }  
        return o.equals(o1);  
    }  
    @Override  
    public int hashCode(Object o) throws HibernateException {  
        return o.hashCode();  
    }  
}
```

```
/**
 * 从JDBC ResultSet读取数据,将其转换为自定义类型后返回
 * (此方法要求对可能出现null值进行处理)
 * names中包含了当前自定义类型的映射字段名称
 * @param resultSet
 * @param names
 * @param owner
 * @return
 * @throws HibernateException
 * @throws SQLException
 */
@Override
public Object nullSafeGet(ResultSet resultSet, String[] names, Object owner) throws HibernateException, SQLException {
    String json = resultSet.getString(names[0]);
    if(json == null || json.trim().length() == 0) {
        return new JsonList();
    }
    return JSONArray.toList(JSONArray.fromObject(json), JsonList.class);
}

/**
 * 本方法将在Hibernate进行数据保存时被调用
 * 我们可以通过PreparedStatement将自定义数据写入到对应的数据库表字段
 * @param preparedStatement
 * @param value
 * @param i
 * @throws HibernateException
 * @throws SQLException
 */
@Override
public void nullSafeSet(PreparedStatement preparedStatement, Object value, int i) throws HibernateException, SQLException {
    if(value == null) {
        preparedStatement.setNull(i, Hibernate.STRING.sqlType());
    } else {
        preparedStatement.setString(i, JSONArray.fromObject(value).toString());
    }
}
```

```
}

/**
 * 提供自定义类型的完全复制方法
 * 本方法将用构造返回对象
 * 当nullSafeGet方法调用之后，我们获得了自定义数据对象，在向用户返回自定义数据之前，
 * deepCopy方法将被调用，它将根据自定义数据对象构造一个完全拷贝，并将此拷贝返回给用户
 * 此时我们就得到了自定义数据对象的两个版本，第一个是从数据库读出的原始版本，其二是我们通过
 * deepCopy方法构造的复制版本，原始的版本将有Hibernate维护，复制版由用户使用。原始版本用作
 * 稍后的脏数据检查依据；Hibernate将在脏数据检查过程中将两个版本的数据进行对比（通过调用
 * equals方法），如果数据发生了变化（equals方法返回false），则执行对应的持久化操作
 *
 * @param o
 * @return
 * @throws HibernateException
 */
@Override
public Object deepCopy(Object o) throws HibernateException {
    if(o == null) return null;
    JsonList jsonList = new JsonList();
    jsonList.addAll((List)o);
    return jsonList;
}

/**
 * 本类型实例是否可变
 * @return
 */
@Override
public boolean isMutable() {
    return true;
}

/* 序列化 */
@Override
public Serializable disassemble(Object value) throws HibernateException {
    return ((Serializable)value);
}
```

```
    }

    /* 反序列化 */
    @Override
    public Object assemble(Serializable cached, Object owner) throws HibernateException {
        return cached;
    }

    @Override
    public Object replace(Object original, Object target, Object owner) throws HibernateException {
        return original;
    }
}
```

JSON框架使用的是json-lib 2.1。

3、自定义JsonList

```
package cn.javass.framework.hibernate;

public class JsonList<T> extends ArrayList implements Cloneable {
}
```

就这么简单，欢迎大家讨论。

订单Model属于公司机密，不方便附上，其他两个类请附件下载（JsonType和JsonList的源代码）。

有人说有性能问题，我写了个测试用例：

测试机器：CPU：p8700(双核@2.53GHZ) 内存：2G

一、插入

1、JSON方式插入10w条

create 100000 elapsed time(millis):21031

2、关联表插入10w条

create 100000 elapsed time(millis):79219

JSON性能远远好于关联表，关联表要插入两个表。

二、查询

1、JSON方式分页（100条一页）查询10w条

select 100000 elapsed time(millis):146047

2、关联表分页（100条一页）查询10w条

select 100000 elapsed time(millis):275375

JSON性能远远好于关联表，关联表需要join连表查询。

JSON方式的缺点：分析统计等查询是鸡肋、大数据量是鸡肋（一列存储数据量不可能太大）。

附件中的 performance.rar是测试用例，欢迎测试拍砖。。

hibernate.rar

////////////////////////////////////

1、用户没有登录：存储到cookie中

2、用户登录后： 存储到数据库中

分析：

购物车是用户绑定的，一个用户一个

购物车最多只允许存50个（因为cookie大小有限）

实现：

1、关联表：购物车-----购物车明细，又是典型一对多，需要频繁操作关联表

2、JSON：只要一张购物车表，用户数据通过JSON存数据库（如 [{productUuid:1, number:2}, {productUuid:2, number:2}]）

使用JSON方式效率肯定是最高的，虽然有转换过程但是在内存中，比读写磁盘肯定快的多。看上边的测试用例。这种方式的好处不需要单独的关联表，省了一张表，何乐而不为。

1.2 Hibernate 关系映射 总结整理

发表时间: 2012-05-11 关键字: hibernate, 关联映射

《Hibernate 关系映射》是我很早之前收集、总结整理的，在此也发上来 希望对大家有用。因为是很早之前写的，不当之处请指正。

一、概念：

关系：名词，事物之间相互作用、相互联系的状态。

关联：名词：表示对象（数据库表）之间的关系；动词：将对象（数据库表）之间通过某种方式联系起来。

映射：将一种形式转化为另一种形式，包括关系。

级联：动词，有关系的双方中操作一方，另一方也将采取一些动作。

值类型：对象不具备数据库同一性，属于一个实体实例其持久化状态被嵌入到所拥有的实体的表行中，没有标识符。

实体类型：具有数据库标识符。

二、数据库：

1、关系

2.1.1、一对一、一对多、多对多

2.1.2、如何表示？外键+索引

2、级联：

2.2.1、级联删除

三、面向对象语言中（Java中）：

1、关系

3.1.1、一对一、一对多、多对多

3.1.2、如何表示？实例变量（对象+集合）

2、级联：

3.2.1、级联删除

3.2.2、级联更新

3.2.3、级联保存

四、如何把数据库关系表示为面向对象中的关系：

1、**关联**：将数据库表之间的关系转化为对象之间的关系；在Hibernate中总指实体之间的关系。

2、**映射**：完成java对象到数据库表的双向转换。

3、**级联（可选）**：将数据库中的级联转化为对象中的级联（两者（对象和数据库表）没关系）。

4、Hibernate的表和对象的映射：

1、实体类型映射：

4.1.1、主键之间的映射

4.1.2、类属性与表字段的映射

4.1.3、组件映射

4.1.4、集合映射

2、实体关联关系映射：

4.2.1、关联关系映射

五、Hibernate映射示例：

5.1、实现

5.1.1、数据库表定义(主表)

5.1.1.1、用户表

```
CREATE TABLE TBL_USER (  
    UUID NUMBER(10) NOT NULL,  
    NAME VARCHAR2(100),  
    AGE NUMBER(10) NOT NULL,  
    PROVINCE VARCHAR2(100),  
    CITY VARCHAR2(100),  
    STREET VARCHAR2(100),  
    CONSTRAINT PK_USER PRIMARY KEY(UUID));
```

5.1.1.2、用户普通信息表(一个用户有一个资料)

```
CREATE TABLE TBL_USER_GENERAL (  
    UUID NUMBER(10) NOT NULL,  
    REALNAME VARCHAR2(10),  
    GENDER VARCHAR2(10),  
    BIRTHDAY NUMBER(10),  
    HEIGHT NUMBER(10),  
    WEIGHT NUMBER(10) ,  
    CONSTRAINT PK_USER_GENERAL PRIMARY KEY(UUID),  
    CONSTRAINT FK_USER_GENERAL FOREIGN KEY(UUID)  
    REFERENCES TBL_USER(UUID));
```

5.1.1.3、农场表(一个用户有多个农场)

```
CREATE TABLE TBL_FARM (  
    UUID NUMBER(10) NOT NULL,  
    NAME VARCHAR2(10),  
    FK_USER_ID NUMBER(10),  
    CONSTRAINT PK_FARM PRIMARY KEY(UUID),  
    CONSTRAINT FK_USER_FARM FOREIGN KEY(FK_USER_ID)  
    REFERENCES TBL_USER(UUID));
```

5.1.2、对象定义

5.1.2.1、用户地址Model

```
package cn.javass.h3test.model;

public class AddressModel implements java.io.Serializable {
    private String province;//省
    private String city;//市
    private String street;//街道
}
```

5.1.2.2、用户Model

```
package cn.javass.h3test.model;

import java.util.HashSet;
import java.util.Set;

public class UserModel implements java.io.Serializable {
    private int uuid;
    private String name;//名称
    private int age;//年龄
    private AddressModel address;//地址
    private UserGeneralModel userGeneral;//用户普通信息
    private Set<FarmModel> farms = new HashSet<FarmModel>();//拥有的农场
}
```

5.1.2.3、用户普通信息Model

```
package cn.javass.h3test.model;

public class UserGeneralModel implements java.io.Serializable {
    private int uuid;
    private String realname;//真实姓名
    private String gender;//性别
    private String birthday;//生日
    private int weight;//体重
    private int height;//身高
    private UserModel user;//所属用户
}
```

5.1.2.4、农场Model

```
package cn.javass.h3test.model;

public class FarmModel implements java.io.Serializable {
```

```
private int uuid;  
private String name;//农场的名称  
private UserModel user;//所属用户  
}
```

5.2、配置

5.2.1、实体类型映射：

5.2.1.1、主键的映射 (UserModel.hbm.xml)

```
<id name="uuid">  
  <generator class="sequence">  
    <param name="sequence">user_uuid</param>  
  </generator>  
</id>
```

5.2.1.2、类属性与表字段的映射 (UserModel.hbm.xml)

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

5.2.1.3、组件映射 (UserModel.hbm.xml)

```
<component name="address" class="cn.javass.h3test.model.AddressModel">  
  <property name="province"/>  
  <property name="city"/>  
  <property name="street"/>  
</component>
```

5.2.1.4、集合映射(Set、List、Map) (都是通过外键连接的 , , , 默认延迟抓取)

Set:

```
private Set<String> farmSet = new HashSet<String>();
```



```
<set name="farmSet" table="TBL_FARM" >
<key column="fk_user_id"/><!--该外键是tbl_farm的-->
    <element type="string" column="name"/>
</set>
```

```
private List<String> farmList = new ArrayList<String>();
```

```
<list name="farmList" table="TBL_FARM">
    <key column="fk_user_id"/>
    <list-index column="uuid"/></list-index>
    <element type="string" column="name"/>
</list>
```

```
private Map<Integer, String> farmMap = new HashMap<Integer, String>();
```

```
<map name="farmMap" table="TBL_FARM">
<key column="fk_user_id"/>
<map-key type="int" column="uuid"/>
<element type="string" column="name"/></element>
</map>
```

对于集合类型默认是延迟加载的，且只能单向导航，不能双向。

5.2.2、实体关联关系映射：

5.2.2.1、单向关联关系映射，不演示。

5.2.2.2、双向关联关系映射

单向

定义：不知道另一端什么情况，获取一端另一端自动获取，因为单向，你不知道另一侧是什么。

如 `class A{ B b;}`

`class B{ }`

只能从A导航到B，不能从B导航到A

关系维护：另一端维护，如B维护

双向

定义：知道另一端（两个单向），从一端获取另一端，从另一端也能获取一端

如 `class A{ B b;}`

`class B{ A a;}`

只能从A导航到B，也能从B导航到A

关系维护：两端，对关联的一侧所作的改变，会立即影响到另一侧

关联的多样性：

从一侧看是多对一，从另一侧看是一对多

另外还有一对一、多对多

EJB CMP：天生双向，对关联的一侧所作的改变，会立即影响到另一侧，

如`UserGeneral.set(user)`，则自动调用`user.setUserGeneral(userGeneral)`

Hibernate、JPA：天生单向，两侧关系的维护是不同的关联，必须手工维护

如`UserGeneral.set(user)`，则需要手工调用`user.setUserGeneral(userGeneral)`。

5.2.2.3、一对一主键关系映射(非延迟抓取)

配置1 (UserModel.hbm.xml)

```
<one-to-one name="userGeneral" cascade="all"/>
```

配置2 (UserGeneralModel.hbm.xml)

```
<id name="uuid">
<generator class="foreign">
    <param name="property">user</param>
</generator>
</id>
<one-to-one name="user"
class="cn.javass.h3test.model.UserModel"/>
```

关联的对象所对应的数据库表之间，通过一个外键引用对主键进行约束。

测试：保存对象，只需保存user，自动级联保存用户信息Model

```
UserModel user = new UserModel();
user.setName("昵称");
UserGeneralModel userGeneral = new UserGeneralModel();
userGeneral.setRealname("真实姓名");
userGeneral.setUser(user);
user.setUserGeneral(userGeneral);
session.save(user);
//若没有cascade="all",这句必须
//session.save(userGeneral);
```

1、一对一必须手工维护双向关系。

2、cascade="all"：表示保存user时自动保存userGeneral，否则还需要一条save (userGeneral)

3、constrained：添加把**userGeneral表的主键映射到**user主键的外键约束

5.2.2.4、一对多关系映射（父/子关系映射）

配置1（UserModel.hbm.xml）

```
<set name="farms" cascade="all">
<key column="fk_user_id"/>
    <one-to-many class="cn.javass.h3test.model.FarmModel"/>
</set>
```

配置2（FarmModel.hbm.xml）

```
<many-to-one name="user" column="fk_user_id"
class="cn.javass.h3test.model.UserModel">
```

测试：保存对象，只需保存user，自动级联保存用户信息Model

```
UserModel user = new UserModel();
user.setName("昵称");
UserGeneralModel userGeneral = new UserGeneralModel();
userGeneral.setRealname("真实姓名");
userGeneral.setUser(user);
user.setUserGeneral(userGeneral);
FarmModel farm = new FarmModel();
farm.setName("farm1");
farm.setUser(user);
user.getFarms().add(farm);
//session.save(farm);//若没有cascade=all的话需要这条语句
session.save(user);
```

以上配置有问题：

```
insert into TBL_USER (name, age, province, city, street, uuid) values (?, ?, ?, ?, ?, ?)

insert into TBL_USER_GENERAL (realname, gender, birthday, weight, height, uuid) values (?, ?, ?, ?, ?, ?)

insert into TBL_FARM (name, fk_user_id, uuid) values (?, ?, ?)

update TBL_FARM set fk_user_id=? where uuid=?
```

1、持久化user(UserModel)；

2、持久化user的一对一关系，即userGeneral(UserGeneralModel)；

3、持久化user的一对多关系，即farms(Set<FarmModel>)；

3.1、首先发现farm是TO，级联save；(因为在这可能是PO，PO的话就应该update，而不是save)；

3.2、其次发现farm在farms集合中，因此需要更新外键 (fk_user_id)，即执行 “update TBL_FARM set fk_user_id=? where uuid=? ”。

解决这个问题：

告诉Hibernate应该只有一端来维护关系（外键），另一端不维护；通过指定<set>端的inverse=” true”，表示关系应该由farm端维护。即更新外键 (fk_user_id) 将由farm端维护。

配置修改 (UserModel.hbm.xml)

```
<set name="farms" cascade="all" inverse="true">
<key column="fk_user_id"/>
```

```
<one-to-many class="cn.javass.h3test.model.FarmModel"/>
</set>
```

再测试：保存对象，只需保存user，自动级联保存用户信息Model

```
UserModel user = new UserModel();
user.setName("昵称");
UserGeneralModel userGeneral = new UserGeneralModel();
userGeneral.setRealname("真实姓名");
userGeneral.setUser(user);
user.setUserGeneral(userGeneral);

FarmModel farm = new FarmModel();
farm.setName("farm1");
farm.setUser(user);
user.getFarms().add(farm);

//session.save(farm);//若没有cascade=all的话需要这条语句
session.save(user);
```

更新外键，需要修改FarmModel的外键并update：

```
insert into TBL_USER (name, age, province, city, street, uuid) values (?, ?, ?, ?, ?, ?)

insert into TBL_USER_GENERAL (realname, gender, birthday, weight, height, uuid) values (?, ?, ?, ?, ?, ?)

insert into TBL_FARM (name, fk_user_id, uuid) values (?, ?, ?)
```

级联删除

1、当删除user时自动删除user下的farm

```
user = (UserModel) session.get(UserModel.class, 1);
session.delete(user);
```

结果：

```
Hibernate: delete from TBL_USER_GENERAL where uuid=?
Hibernate: delete from TBL_FARM where uuid=?
Hibernate: delete from TBL_USER where uuid=?
```

2、删除user中的farms的一个元素

```
UserModel user =
(UserModel) session.get(UserModel.class, 118);
FarmModel farm = (FarmModel) user.getFarms().toArray()[user.getFarms().size() - 1];
user.getFarms().remove(farm); //1.必须先从集合删除
session.delete(farm); //2.然后才能删除
```

结果：

```
Hibernate: delete from TBL_FARM where uuid=?
```

如果将子对象从集合中移除，实际上我们是想删除它。要实现这种要求，就必须使用`cascade="all-delete-orphan"`。无需再调用`session.delete(farm)`

5.2.2.5、多对多关系映射：不用

为什么不使用多对多：当添加新字段时给谁？

那实际项目如何用：拆成两个一对多。

六、涉及的SQL语句会按照下面的顺序发出执行：

1、查询

1、所有对实体进行插入的语句，其顺序按照对象执行Session.save()的时间顺序

2、所有对实体进行更新的语句

3、所有进行集合插入的语句（实体类型）

4、所有对集合元素进行删除、更新或插入的语句（值类型）

5、所有进行集合删除的语句（实体类型）

6、所有对实体进行删除的语句，其顺序按照对象执行Session.delete()的时间顺序

（有一个例外是，如果对象使用native方式来生成ID（持久化标识）的话，它们一执行save就会被插入。）

七、影响关系映射抓取的cfg配置：

hibernate.max_fetch_depth

为单向关联(一对一, 多对一)的外连接抓取 (outer join fetch) 树设置最大深度. 值为0意味着将关闭默认的外连接抓取.

取值 建议在0到3之间取值

为Hibernate关联的批量抓取设置默认数量.

hibernate.default_batch_fetch_size

取值 建议的取值为4, 8, 和16

如果你的数据库支持ANSI, Oracle或Sybase风格的外连接, 外连接抓取通常能通过限制往返数据库次数 (更多的工作交由数据库自己来完成)来提高效率. 外连接抓取允许在单个SELECTSQL语句中, 通过many-to-one, one-to-many, many-to-many和one-to-one关联获取连接对象的整个对象图.

将hibernate.max_fetch_depth设为0能在全局 范围内禁止外连接抓取. 设为1或更高值能启用one-to-one和many-to-one外连接关联的外连接抓取, 它们通过 fetch="join"来映射.

八、抓取策略

1、抓取策略定义

抓取策略 (fetching strategy) 是指：当应用程序需要在 (Hibernate实体对象图的) 关联关系间进行导航的时候， Hibernate如何获取关联对象的策略。抓取策略可以在O/R映射的元数据中声明，也可以在特定的HQL 或条件查询 (Criteria Query) 中重载声明。

2、Hibernate3 定义了如下几种抓取策略：

连接抓取 (Join fetching) - Hibernate通过 在SELECT语句使用OUTER JOIN (外连接) 来 获得**对象**的关联实例或者关联集合。默认非延迟加载

集合抓取需要通过配置fetch="join"来指定。下行数据太多 (冗余) , IO

```
//配置 fetch="join"( lazy="true"不起作用了)
session.get(UserModel.class, 118);//是获取对象的
Hibernate: select ... from TBL_USER userModel0_, TBL_FARM farms1_
where userModel0_.uuid=farms1_.fk_user_id(+) and userModel0_.uuid=?
```

查询抓取 (Select fetching) - 另外发送一条 SELECT 语句抓取当前**对象**的关联实体或集合。除非你显式的指定lazy="false"禁止延迟抓取 (lazy fetching) , 否则只有当你真正访问关联关系的时候, 才会执行第二条select语句。

```
////配置 lazy=" true" 默认 ( 或者lazy="false" fetch="select" )
session.get(UserModel.class, 118);//是获取对象的
Hibernate: select ... from TBL_USER usermodel0_ where usermodel0_.uuid=?
Hibernate: select ... from TBL_FARM farms0_ where farms0_.fk_user_id=?
```

默认用于lazy="true"情况的集合抓取, 如果lazy="false", 需要指定fetch="select"来通过查询抓取。会造成DB的CPU利用率非常高, 计算密集

子查询抓取 (Subselect fetching) - 另外发送一条SELECT 语句抓取在前面查询到 (或者抓取到) 的所有实体对象的关联集合。除非你显式的指定lazy="false" 禁止延迟抓取 (lazy fetching) , 否则只有当你真正访问关联关系的时候, 才会执行第二条select语句。

当通过Query等接口查询多个实体时, 如果指定**fetch="subselect"**则将通过子查询获取集合

```
////配置fetch="subselect"
Query q = session.createQuery("from UserModel");
System.out.println(q.list());
Hibernate: select ..... from TBL_USER usermodel0_
Hibernate: select ..... from TBL_FARM farms0_ where farms0_.fk_user_id
in (select usermodel0_.uuid from TBL_USER usermodel0_)
```

批量抓取 (Batch fetching) - 对查询抓取的优化方案, 通过指定一个主键或外键列表, Hibernate使用单条SELECT语句获取一批对象实例或集合。

当通过Query等接口查询多个实体时, 如果指定farm的**batch-size="....."**则将通过使用单条SELECT语句获取一批对象实例或集合。

```
Query q = session.createQuery("from UserModel");  
List<UserModel> userList = q.list();          System.out.println(userList);  
Hibernate: select ... TBL_USER usermodel0_  
Hibernate: select ... from TBL_FARM farms0_ where farms0_.fk_user_id in (?, ?)
```

可指定全局批量抓取策略：hibernate.default_batch_fetch_size，取值：建议的取值为4, 8, 和16。

如果batch-size="4"，而某个user有19个农场，Hibernate将只需要执行五次查询，分别为4、4、4、4、3。

测试必须数据量足够多，，如果只有一条不行

3、使用延迟属性抓取（Using lazy property fetching）

属性的延迟载入要求在其代码构建时加入二进制指示指令（bytecode instrumentation），如果你的持久类代码中未含有这些指令，Hibernate将会忽略这些属性的延迟设置，仍然将其直接载入。

Hibernate3对单独的属性支持延迟抓取，这项优化技术也被称为组抓取（fetch groups）。请注意，该技术更多的属于市场特性。在实际应用中，优化行读取比优化列读取更重要。但是，仅载入类的部分属性在某些特定情况下会有用，例如在原有表中拥有几百列数据、数据模型无法改动的情况下。

4、Hibernate在抓取时会lazy区分下列各种情况：

立即抓取 - 当宿主被加载时，关联、集合或属性被立即抓取。

Lazy collection fetching , 延迟集合抓取- 直到应用程序对集合进行了一次操作时，集合才被抓取。（对集合而言这是默认行为。）

Extra-lazy" collection fetching,"Extra-lazy"集合抓取 -对集合类中的每个元素而言，都是直到需要时才去访问数据库。除非绝对必要，Hibernate不会试图去把整个集合都抓取到内存里来（适用于非常大的集合）。

```
// lazy="extra"
Query q = session.createQuery("from UserModel");
Iterator it = q.iterate();          System.out.println(((UserModel)it.next()).getFarms().size());
//或
List<UserModel> userList = q.list();      System.out.println(userList.get(0).getFarms().size());

Hibernate: select userModel0_.uuid as col_0_0_ from TBL_USER userModel0_
Hibernate: select ... from TBL_USER userModel0_ where userModel0_.uuid=?
Hibernate: select count(uuid) from TBL_FARM where fk_user_id =?
//或
Hibernate: select ... from TBL_USER userModel0_
Hibernate: select count(uuid) from TBL_FARM where fk_user_id =?
```

对于调用size()、contains、isEmpty是一种优化，不读取所有级联，而是按条件生产不同的sql。

Proxy fetching , 代理抓取 - 对返回单值的关联而言，当其某个方法被调用，而非对其关键字进行get操作时才抓取。

```
//默认 <many-to-one name="user" .....lazy="false"/>
FarmModel farm = (FarmModel) session.get(FarmModel.class, 121);
System.out.println(farm.getUser().getUuid());

Hibernate: select ... from TBL_FARM farmmodel0_ where farmmodel0_.uuid=?
Hibernate: select ... from TBL_USER userModel0_ where userModel0_.uuid=?
118
```

```
// <many-to-one name="user" .....lazy="proxy"/>
FarmModel farm = (FarmModel) session.get(FarmModel.class, 121);
System.out.println(farm.getUser().getUuid());

Hibernate: select ... from TBL_FARM farmmodel0_ where farmmodel0_.uuid=?
118
```

注：如果constrained="false"或基于主键的一对一，不可能使用代理，Hibernate会采取预先抓取！

"No-proxy" fetching,非代理抓取 - 对返回单值的关联而言，当实例变量被访问的时候进行抓取。与上面的代理抓取相比，这种方法没有那么“延迟”得厉害(就算只访问标识符，也会导致关联抓取)但是更加透明，因为对应用程序来说，不再看到proxy。这种方法需要在编译期间进行字节码增强操作，因此很少需要用到。

Lazy attribute fetching，属性延迟加载 - 对属性或返回单值的关联而言，当其实例变量被访问的时候进行抓取。需要编译期字节码强化，因此这一方法很少是必要的。

这里有两个正交的概念：关联何时被抓取，以及被如何抓取（会采用什么样的SQL语句）。不要混淆它们！我们使用抓取来改善性能。我们使用延迟来定义一些契约，对某特定类的某个脱管的实例，知道有哪些数据是可以使用的。

九、抓取优化

1、集合N+1：

可以使用batch-size来减少获取次数，即如batch-size=" 10"，则是N/10+1。

开启二级缓存。

对于集合比较小且一定会用到的可采用fetch=" join"，这样只需一条语句。

2、笛卡尔积问题：

```
<set name="farms" cascade="all,all-delete-orphan" inverse="true" fetch="join">
<key column="fk_user_id"/>
  <one-to-many class="cn.javass.h3test.model.FarmModel"/>
</set>
<set name="hourses" cascade="all,all-delete-orphan" inverse="true" fetch="join">
<key column="fk_user_id"/>
  <one-to-many class="cn.javass.h3test.model.HourseModel"/>
</set>
```

如上配置产生笛卡尔积问题。

```
select user.*,farm.*,hourse.* from UserModel user, FarmModel farm, HourseModel hourse
where user.uuid=farm.fk_user.uuid(+) and
      user.uuid=hourse.fk_user.uuid(+)
```

解决方案：

- 1、**fetch=" subselect" ，子查询，每个User查询一套笛卡尔积**
- 2、**完全不采用关系映射。**
- 3、**大集合采用批处理，按块获取集合数据**
- 4、**复杂SQL太复杂太慢：找DBA优化，索引等是否有效，是否加载了过多的无用数据，拆分SQL，按需获取数据。**
- 5、**按需获取1对多中的集合。**
- 6、**缓存**
-

不当之处，请多多指正！

1.3 Hibernate 二级缓存 总结整理

发表时间: 2012-05-13 关键字: hibernate

和《[Hibernate 关系映射 收集、总结整理](#)》一样，本篇文章也是我很早之前收集、总结整理的，在此也发上来 希望对大家有用。因为是很早之前写的，不当之处请指正。

1、缓存：缓存是什么，解决什么问题？

位于速度相差较大的两种硬件/软件之间的，用于协调两者数据传输速度差异的结构，均可称之为Cache（摘自Robbin的《[缓存技术浅谈](#)》）。目的：让数据更接近于应用程序，协调速度不匹配，使访问速度更快。（请参考<http://baike.baidu.com/view/907.htm> 了解更多缓存知识）

高速缓存不属于Hibernate等，属于独立产品或框架，可单独使用。

常见缓存算法：

- a) **LFU (Least Frequently Used)**：最近不常被使用（命中率低），一定时间段内使用次数最少的
- b) **LRU (Least Recently Used)**：最近很少使用（LinkedHashMap），没有被使用时间最长的
- c) **FIFO (First In First Out)**：先进先出

2、缓存策略

- 1．对象缓存
- 2．查询缓存
- 3．页面缓存
 - 1．动态页面缓存
 - 2．Servlet缓存
 - 3．页面片段缓存

3、缓存分类

1. Web缓存：
 - i. 浏览器缓存：ajax（在客户端缓存）、HTTP协议
 - ii. 代理服务器缓存
2. 操作系统缓存：如用于减少磁盘操作
3. 数据库缓存：
 - i. 结果缓存：
 - ii. 排序缓存
 - iii. 插入缓存
 - iv. 日志缓存
 - v.
4. 应用程序缓存
 - i. 对象缓存
 - ii. 查询缓存
 - iii. 页面缓存
 1. 动态页面静态化：网页静态化、独立图片服务器
 2. 页面局部缓存：
 3. 请求回应缓存：

4、常见Java缓存框架

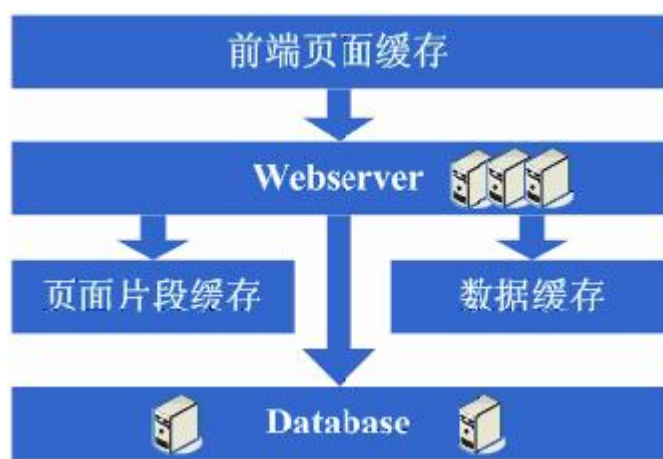
- § EHCACHE
- § OSCACHE
- § JBOSSCACHE
- § SWARMCACHE

5、通用缓存产品

§ Memcached：在大规模互联网应用下使用，可用于分布式环境，每秒支撑1.5万~2万次请求

§ Tokyo Tyrant：兼容memcached协议，可以持久化存储，支持故障切换，对缓存服务器有高可靠性要求可以使用，每秒支撑0.5万~0.8万次请求

6、基于Web应用的缓存应用场景：



（摘自bluedavy的[《大型网站架构演化》](#)）

8、缓存实战：

8.4、ORM缓存

8.4.1、目的：

Hibernate缓存：使当前数据库状态的表示接近应用程序，要么在内存中，要么在应用程序服务器机器的磁盘上。高速缓存是数据的一个本地副本，处于应用程序和数据库之间，可用来避免数据库的命中。

8.4.2、避免数据库命中：

应用程序根据标识符到缓存查，有就返回，没有再去数据库。

8.4.3、ORM缓存分类

一级缓存、二级缓存

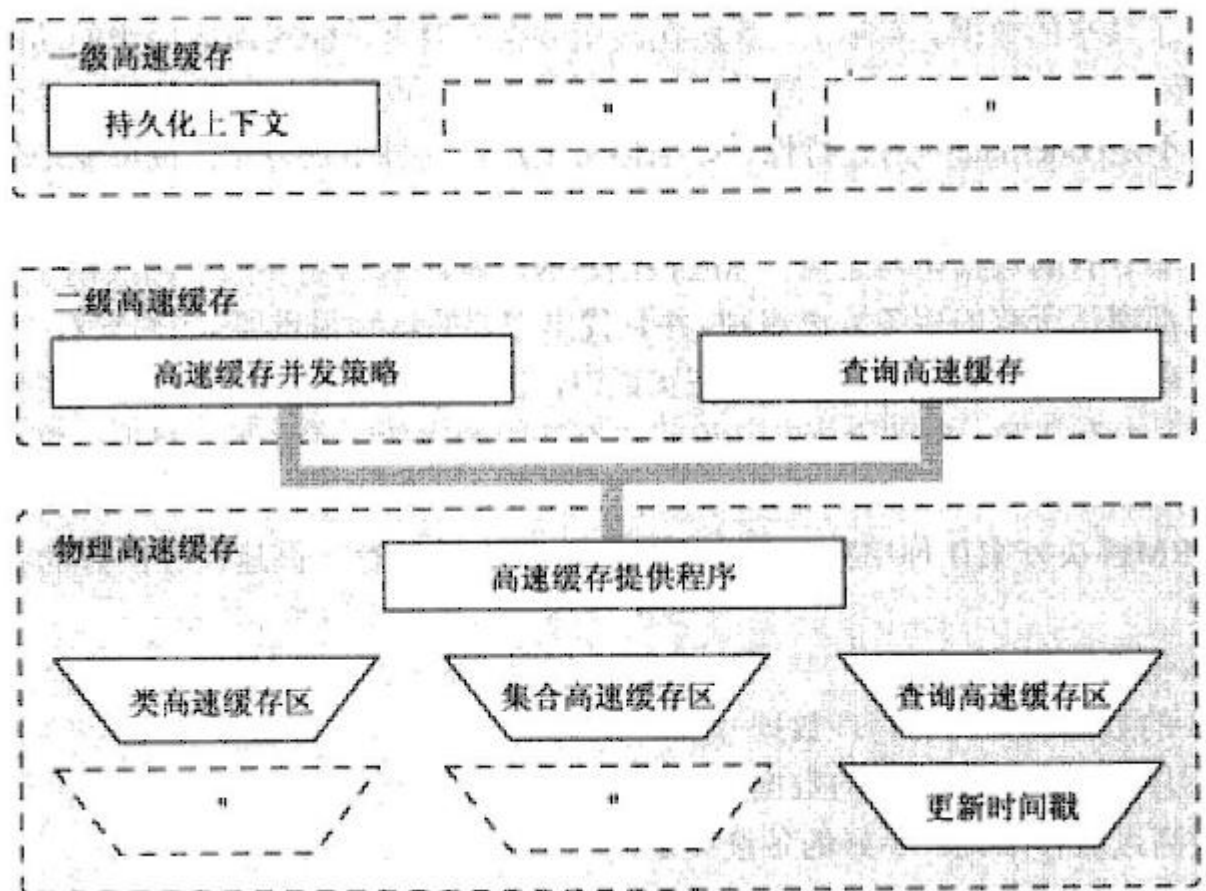
8.4.4、缓存范围

- 1、事务范围高速缓存，对应于一级缓存（单Session）
- 2、过程（JVM）范围高速缓存，对应于二级缓存（单SessionFactory）
- 3、集群范围高速缓存，对应于二级缓存（多SessionFactory）

8.4.5、缓存哪些数据

- 1、很少改变的数据；
- 2、不重要的数据，如论坛帖子，无需实时的数据；
- 3、应用程序固有的而非共享的。
- 4、读大于写有用

8.4.6、Hibernate缓存架构



图摘自《Hibernate in Action》

- § Hibernate中的二级缓存是可插拔的。
- § Hibernate二级缓存支持对象缓存、集合缓存、查询结果集缓存，对于查询结果集缓存可选。
- § 查询缓存：需要两个额外的物理高速缓存区域：一个用于存放查询的结果集；另一个用于存储表上次更新的时间戳

各种缓存提供商对缓存并发策略的支持情况

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss Cache 1.x	yes			yes
JBoss Cache 2	yes			yes

8.4.6.2、高速缓存实战(ehcache)

8.4.6.2.1、全局配置(hibernate.cfg.xml)

```
<!-- 开启二级缓存 -->
<property name="hibernate.cache.use_second_level_cache">true</property>
<!-- 开启查询缓存 -->
<property name="hibernate.cache.use_query_cache">true</property>
<!-- 二级缓存区域名的前缀 -->
<!--<property name="hibernate.cache.region_prefix">h3test</property>-->
<!-- 高速缓存提供程序 -->
<property name="hibernate.cache.region.factory_class">
net.sf.ehcache.hibernate.EhCacheRegionFactory
</property>
<!-- 指定缓存配置文件位置 -->
<property name="hibernate.cache.provider_configuration_file_resource_path">
ehcache.xml
</property>
<!-- 强制Hibernate以更人性化的格式将数据存入二级缓存 -->
```

```
<property name="hibernate.cache.use_structured_entries">true</property>

<!-- Hibernate将收集有助于性能调节的统计数据 -->
<property name="hibernate.generate_statistics">true</property>
```

8.4.6.2.2、ehcache配置 (ehcache.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache name="h3test">
    <defaultCache
        maxElementsInMemory="100"
        eternal="false"
        timeToIdleSeconds="1200"
        timeToLiveSeconds="1200"
        overflowToDisk="false">
    </defaultCache>
</ehcache>
```

8.4.6.2.3、实体只读缓存

1、修改FarmModel.hbm.xml,添加如下红色部分配置，表示实体缓存并只读

```
<hibernate-mapping>
    <class name="cn.javass.h3test.model.FarmModel" table="TBL_FARM">
        <cache usage="read-only"/>
        .....
    </class>
</hibernate-mapping>
```

2、测试代码

```
public static void readonlyTest() {
    SessionFactory sf =
new Configuration().configure().buildSessionFactory();
```

```
Session session1 = sf.openSession();
Transaction t1 = session1.beginTransaction();
//确保数据库中有标识符为1的FarmModel
FarmModel farm = (FarmModel) session1.get(FarmModel.class, 1);
//如果修改将报错，只读缓存不允许修改
//farm.setName("aaa");
t1.commit();
session1.close();

Session session2 = sf.openSession();
Transaction t2 = session2.beginTransaction();

farm = (FarmModel) session2.get(FarmModel.class, 1);

t2.commit();
session2.close();
sf.close();
}
```

§ 只读缓存不允许更新，将报错Can't write to a readonly object。

§ 允许新增，（现在2.0新增直接添加到二级缓存）

8.4.6.2.4、实体非严格读/写缓存

1、修改FarmModel.hbm.xml,添加如下红色部分配置，表示实体缓存并非严格读/写

```
<hibernate-mapping>
    <class name="cn.javass.h3test.model.FarmModel" table="TBL_FARM">
        <cache usage="nonstrict-read-write"/>
        .....
    </hibernate-mapping>
```

2、测试代码

```
public static void nonstrictReadWriteTest () {  
    SessionFactory sf =  
new Configuration().configure().buildSessionFactory();  
  
    Session session1 = sf.openSession();  
    Transaction t1 = session1.beginTransaction();  
    //确保数据库中有标识符为1的FarmModel  
    FarmModel farm = (FarmModel) session1.get(FarmModel.class, 1);  
    t1.commit();  
    session1.close();  
  
    Session session2 = sf.openSession();  
    Transaction t2 = session2.beginTransaction();  
  
    farm = (FarmModel) session2.get(FarmModel.class, 1);  
  
    t2.commit();  
    session2.close();  
    sf.close();  
}
```

§ 允许更新，更新后缓存失效，需再查询一次。

§ 允许新增，新增记录自动加到二级缓存中。

§ 整个过程不加锁，不保证。

8.4.6.2.5、实体读/写缓存

1、修改FarmModel.hbm.xml,添加如下红色部分配置，表示实体缓存并读/写

```
<hibernate-mapping>  
    <class name="cn.javass.h3test.model.FarmModel" table="TBL_FARM">  
        <cache usage="read-write"/>  
    </class>  
</hibernate-mapping>
```

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

2、测试代码

```
public static void readWriteTest() {  
    SessionFactory sf =  
new Configuration().configure().buildSessionFactory();  
  
    Session session1 = sf.openSession();  
    Transaction t1 = session1.beginTransaction();  
    //确保数据库中有标识符为1的FarmModel  
    FarmModel farm = (FarmModel) session1.get(FarmModel.class, 1);  
    farm.setName("as");  
    t1.commit();  
    session1.close();  
  
    Session session2 = sf.openSession();  
    Transaction t2 = session2.beginTransaction();  
    farm = (FarmModel) session2.get(FarmModel.class, 1);  
    t2.commit();  
    session2.close();  
    sf.close();  
}
```

§ 允许更新，更新后自动同步到缓存。

§ 允许新增，新增记录后自动同步到缓存。

§ 保证Read committed隔离级别及可重复读隔离级别（通过时间戳实现）

§ 整个过程加锁，如果当前事务的时间戳早于二级缓存中的条目的时间戳，说明该条目已经被别的事务修改了，此时重新查询一次数据库，否则才使用缓存数据，因此保证可重复读隔离级别。

8.4.6.2.6、实体事务缓存

需要特定缓存的支持和JTA事务支持，此处不演示。

8.4.6.2.7、集合缓存

此处演示读/写缓存示例，其他自测

1、修改FarmModel.hbm.xml,添加如下红色部分配置，表示实体缓存并读/写

```
<hibernate-mapping>
    <class name="cn.javass.h3test.model.UserModel" table="TBL_USER">
        <cache usage="read-write" />
        <set name="farms" cascade="all" inverse="true" lazy="false">
            <cache usage="read-write"/>
            <key column="fk_user_id"/>
            <one-to-many class="cn.javass.h3test.model.FarmModel"/>
        </set>
    </class>
</hibernate-mapping>
```

2、测试代码

```
public static void collectionReadWriteTest() {
    SessionFactory sf =
    new Configuration().configure().buildSessionFactory();

    Session session1 = sf.openSession();
    Transaction t1 = session1.beginTransaction();
    //确保数据库中有标识符为118的UserModel
    UserModel user = (UserModel) session1.get(UserModel.class, 118);
    user.getFarms();
    t1.commit();
    session1.close();

    Session session2 = sf.openSession();
    Transaction t2 = session2.beginTransaction();
    user = (UserModel) session2.get(UserModel.class, 118);
```



```
user.getFarms();
t2.commit();
session2.close();
sf.close();
}
```

§ 和实体并发策略有相同含义；

§ 但集合缓存只缓存集合元素的标识符，在二级缓存中只存放相应实体的标识符，然后再通过标识符去二级缓存查找相应的实体最后组合为集合返回。

8.4.6.2.8、查询缓存

1、保证全局配置中有开启了查询缓存。

2、修改FarmModel.hbm.xml,添加如下红色部分配置，表示实体缓存并读/写

```
<hibernate-mapping>
    <class name="cn.javass.h3test.model.FarmModel" table="TBL_FARM">
        <cache usage="read-write"/>
        .....
    </hibernate-mapping>
```

3、测试代码

```
public static void queryCacheTest() {
    SessionFactory sf =
    new Configuration().configure().buildSessionFactory();

    Session session1 = sf.openSession();
    Transaction t1 = session1.beginTransaction();
    Query query = session1.createQuery("from FarmModel");
    //即使全局打开了查询缓存，此处也是必须的
    query.setCacheable(true);
    List<FarmModel> farmList = query.list();
    t1.commit();
}
```

```
session1.close();

Session session2 = sf.openSession();
Transaction t2 = session2.beginTransaction();
query = session2.createQuery("from FarmModel");
//即使全局打开了查询缓存，此处也是必须的
query.setCacheable(true);
farmList = query.list();
t2.commit();
session2.close();
    sf.close();
}
```

§ 和实体并发策略有相同含义；

§ 和集合缓存类似，只缓存集合元素的标识符，在二级缓存中只存放相应实体的标识符，然后再通过标识符去二级缓存查找相应的实体最后组合为集合返回。

8.4.6.2.9、高速缓存区域

Hibernate在不同的高速缓存区域保存不同的类（实体）/集合，如果不配置区域默认都保存到“默认缓存”（defaultCache）中。

每一个区域可以设置过期策略、缓存条目大小等等。

对于类缓存，默认区域名是全限定类名，如cn.javass.h3test.model.UserModel。

对于集合而言，默认区域名是全限定类名+属性名，如cn.javass.....UserModel.farms。

可通过hibernate.cache.region_prefix指定特定SessionFactory的区域前缀，如前缀是h3test，则如类缓存的区域名就是h3test.cn.javass.h3test.model.UserModel。如果应用程序使用多个SessionFactory这可能是必须的。

可通过<cache usage="read-write" region="区域名"/>自定义区域名，不过默认其实就可以了。

8.4.6.2.10、ehcache配置详解：

1、默认cache：如果没有对应的特定区域的缓存，就使用默认缓存。

```
<defaultCache
    maxElementsInMemory="100"
    eternal="false"
    timeToIdleSeconds="1200"
    timeToLiveSeconds="1200"
    overflowToDisk="false">
</defaultCache>
```

2、指定区域cache：通过name指定，name对应到Hibernate中的区域名即可。

```
<cache name="cn.javass.h3test.model.UserModel"
    maxElementsInMemory="100"
    eternal="false"
    timeToIdleSeconds="1200"
    timeToLiveSeconds="1200"
    overflowToDisk="false">
</cache>
```

3、cache参数详解：

name:指定区域名

maxElementsInMemory：缓存在内存中的最大数目

maxElementsOnDisk：缓存在磁盘上的最大数目

eternal：缓存是否持久

overflowToDisk：硬盘溢出数目

timeToIdleSeconds：当缓存条目闲置n秒后销毁

timeToLiveSeconds：当缓存条目存活n秒后销毁

memoryStoreEvictionPolicy:缓存算法，有LRU（默认）、LFU、FIFO

4、StandardQueryCache

用于查询缓存使用，如果指定了该缓存，那么查询缓存将放在该缓存中。

```
<cache
    name="org.hibernate.cache.StandardQueryCache"
    maxElementsInMemory="5"
    eternal="false"
    timeToLiveSeconds="120"
    overflowToDisk="true"/>
```

如果不给查询设置区域名默认缓存到这，可以通过 “`query.setCacheRegion("区域名");`” 来设置查询的区域名。

5、UpdateTimestampsCache

时间戳缓存，内部使用，用于保存最近更新的表的时间戳，这是非常重要的，无需失效，关闭时间戳缓存区域的过期时间。

```
<cache
    name="org.hibernate.cache.UpdateTimestampsCache"
    maxElementsInMemory="5000"
    eternal="true"
    overflowToDisk="true"/>
```

Hibernate使用时间戳区域来决定被高速缓存的查询结果集是否是失效的。当你重新执行了一个启用了高速缓存的查询时，Hibernate就在时间戳缓存中查找对被查询的（几张）表所做的最近插入、更新或删除的时间戳。如果找到的时间戳晚于高速缓存查询结果的时间戳，那么缓存结果将被丢弃，重新执行一次查询。

8.4.6.2.11、什么时候需要查询缓存

大多数时候无法从结果集高速缓存获益。必须知道:每隔多久重复执行同一查询。

对于那些查询非常多但插入、删除、更新非常少的应用程序来说, 查询缓存可提升性能。但写入多查询少的没有用, 总失效。

8.4.6.2.12、管理一级缓存

无论何时, 当你给save()、update()或 saveOrUpdate()方法传递一个对象时, 或使用load()、 get()、list()、iterate() 或scroll()方法获得一个对象时, 该对象都将被加入到Session的内部缓存中。

当随后flush()方法被调用时, 对象的状态会和数据库取得同步。如果你不希望此同步操作发生, 或者你正处理大量对象、需要对有效管理内存时, 你可以调用evict() 方法, 从一级缓存中去掉这些对象及其集合。

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

Session还提供了一个contains()方法, 用来判断某个实例是否处于当前session的缓存中。

如若要把所有的对象从session缓存中彻底清除, 则需要调用Session.clear()。

CacheMode参数用于控制具体的Session如何与二级缓存进行交互。

CacheMode.NORMAL - 从二级缓存中读、写数据。

CacheMode.GET - 从二级缓存中读取数据, 仅在数据更新时对二级缓存写数据。

CacheMode.PUT - 仅向二级缓存写数据, 但不从二级缓存中读数据。

CacheMode.REFRESH - 仅向二级缓存写数据, 但不从二级缓存中读数据。通过hibernate.cache.use_minimal_puts的设置, 强制二级缓存从数据库中读取数据, 刷新缓存内容。

8.4.6.2.12、管理二级缓存

对于二级缓存来说, 在SessionFactory中定义了许多方法, 清除缓存中实例、整个类、集合实例或者整个集合。

sessionFactory.evict(Cat.class, catId); //evict a particular Cat

sessionFactory.evict(Cat.class); //evict all Cats

sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens

sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections

sessionFactory.evictQueries();//evict all queries

8.4.6.2.13、监控二级缓存

如若需要查看二级缓存或查询缓存区域的内容，你可以使用统计（Statistics）API。

通过sessionFactory.getStatistics()；获取Hibernate统计信息。

此时，你必须手工打开统计选项。

hibernate.generate_statistics true

hibernate.cache.use_structured_entries true

具体详见“hibernate监控.rar”（需要自己稍微改改才能用）

需要修改head.jsp中的如下代码获取sessionFactory

```
if(sessionFactory == null) {  
    WebApplicationContext applicationContext = WebApplicationContextUtils.getWebApplicationContext(  
        sessionFactory = (SessionFactory)applicationContext.getBean("userSessionFactory");  
}
```

参考资料:

Robbin的《[缓存技术浅谈](#)》

百度百科的高速缓存知识 <http://baike.baidu.com/view/907.htm>

bluedavy的《[大型网站架构演化](#)》（<http://www.blogjava.net/BlueDavy/archive/2008/09/03/226749.html>）

《Hibernate in Action》

附件下载:

- hibernate监控.rar (10.7 KB)
- dl.iteye.com/topics/download/b0776b5e-28fc-39dd-b136-7b3819125d0d

1.4 Hibernate自定义类型 对象--->序列化为字符串 存储

发表时间: 2013-04-21

在有些时候 我们可能序列化存储对象为字符串形式，比如会话序列化存储到数据库。（当然数据量小没问题 大了还是存如memcached这种缓存中）。

具体代码：

```
/**
 * Copyright (c) 2005-2012 https://github.com/zhangkaitao
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 */
package com.sishuok.es.common.repository.hibernate.type;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.google.common.collect.Maps;
import org.apache.commons.codec.binary.Hex;
import org.apache.shiro.web.util.SavedRequest;
import org.hibernate.HibernateException;
import org.hibernate.engine.spi.SessionImplementor;
import org.hibernate.usertype.UserType;

import java.io.*;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;
import java.util.HashMap;
import java.util.Map;

/**
 * Object序列化/反序列化
 * 数据库中以hex字符串存储
 */
```



```
* 参考http://jinnianshilongnian.iteye.com/blog/1497791
* <p>User: Zhang Kaitao
* <p>Date: 13-3-20 下午4:46
* <p>Version: 1.0
*/

public class ObjectSerializeUserType implements UserType, Serializable {

    @Override
    public int[] sqlTypes() {
        return new int[] {Types.VARCHAR};
    }

    @Override
    public Class returnedClass() {
        return Object.class;
    }

    @Override
    public boolean equals(Object o, Object o1) throws HibernateException {
        if (o == o1) {
            return true;
        }
        if (o == null || o1 == null) {
            return false;
        }

        return o.equals(o1);
    }

    @Override
    public int hashCode(Object o) throws HibernateException {
        return o.hashCode();
    }

    /**
     * 从JDBC ResultSet读取数据,将其转换为自定义类型后返回
     */
}
```

```
* (此方法要求对可能出现的null值进行处理)
* names中包含了当前自定义类型的映射字段名称
* @param names
* @param owner
* @return
* @throws HibernateException
* @throws SQLException
*/

@Override
public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object
    ObjectInputStream ois = null;
    try {
        String hexStr = rs.getString(names[0]);
        ois = new ObjectInputStream(new ByteArrayInputStream(Hex.decodeHex(hexStr.toCharArray())
        return ois.readObject();
    } catch (Exception e) {
        throw new HibernateException(e);
    } finally {
        try {
            ois.close();
        } catch (IOException e) {
        }
    }
}

/**
 * 本方法将在Hibernate进行数据保存时被调用
 * 我们可以通过PreparedStatement将自定义数据写入到对应的数据库表字段
 */

@Override
public void nullSafeSet(PreparedStatement st, Object value, int index, SessionImplementor s
    ObjectOutputStream oos = null;
    if(value == null) {
        st.setNull(index, Types.VARCHAR);
    } else {
        try {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
```

```
        oos = new ObjectOutputStream(bos);
        oos.writeObject(value);
        oos.close();

        byte[] objectBytes = bos.toByteArray();
        String hexStr = Hex.encodeHexString(objectBytes);

        st.setString(index, hexStr);
    } catch (Exception e) {
        throw new HibernateException(e);
    } finally {
        try {
            oos.close();
        } catch (IOException e) {
        }
    }
}
}
```

/**

* 提供自定义类型的完全复制方法

* 本方法将用构造返回对象

* 当nullSafeGet方法调用之后，我们获得了自定义数据对象，在向用户返回自定义数据之前，

* deepCopy方法将被调用，它将根据自定义数据对象构造一个完全拷贝，并将此拷贝返回给用户

* 此时我们就得到了自定义数据对象的两个版本，第一个是从数据库读出的原始版本，其二是我们通过

* deepCopy方法构造的复制版本，原始的版本将有Hibernate维护，复制版由用户使用。原始版本用作

* 稍后的脏数据检查依据；Hibernate将在脏数据检查过程中将两个版本的数据进行对比（通过调用

* equals方法），如果数据发生了变化（equals方法返回false），则执行对应的持久化操作

*

* @param o

* @return

* @throws HibernateException

*/

@Override

public Object deepCopy(Object o) throws HibernateException {

```
        if(o == null) return null;
        return o;
    }

    /**
     * 本类型实例是否可变
     * @return
     */
    @Override
    public boolean isMutable() {
        return true;
    }

    /* 序列化 */
    @Override
    public Serializable disassemble(Object value) throws HibernateException {
        return ((Serializable)value);
    }

    /* 反序列化 */
    @Override
    public Object assemble(Serializable cached, Object owner) throws HibernateException {
        return cached;
    }

    @Override
    public Object replace(Object original, Object target, Object owner) throws HibernateException {
        return original;
    }
}
```

1、deepCopy 直接返回我们的对象 因为我们修改会话后会直接update下 所以此处没必要复制

2、Hex.encodeHexString ????????16?????

使用代码：

```
@Type(type = "com.sishuok.es.common.repository.hibernate.type.ObjectSerializeUserType")  
private OnlineSession session;
```

[github](#)

1.5 Hibernate自定义类型 集合--->字符串 存储

发表时间: 2013-04-21

场景：

角色[1]-----[*](资源[1]---[*]权限)

某个角色 具有 某个资源的 某些权限，当然此处都是多对多 为了好理解 暂时1---*。这里是资源-对应-多个权限，但是权限一般不会很多，而且我们一般也不会根据权限去查找，因此没必要做个关联表，此处我们可以使用字符串如1,2,3,4来存储其id，这样可以有效减少中间表数量 提高效率。

方案：

如果不想在程序中拼接这种字符串 我们可以考虑使用Hibernate自定义数据类型；即把集合类型--->某个分隔符连接的字符串

```
/**
 * Copyright (c) 2005-2012 https://github.com/zhangkaitao
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 */
package com.sishuok.es.common.repository.hibernate.type;

import com.google.common.collect.Lists;
import org.apache.commons.beanutils.ConvertUtils;
import org.apache.commons.lang3.StringUtils;
import org.hibernate.HibernateException;
import org.hibernate.engine.spi.SessionImplementor;
import org.hibernate.usertype.EnhancedUserType;
import org.hibernate.usertype.ParameterizedType;
import org.hibernate.usertype.UserType;

import java.io.*;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```
import java.sql.SQLException;
import java.sql.Types;
import java.util.Collection;
import java.util.List;
import java.util.Properties;

/**
 * 将List转换为指定分隔符分隔的字符串存储 List的元素类型只支持常见的数据类型 可参考{@link org.apache.
 * <p>User: Zhang Kaitao
 * <p>Date: 13-4-16 上午8:32
 * <p>Version: 1.0
 */
public class CollectionToStringUserType implements UserType, ParameterizedType, Serializable {

    /**
     * 默认,
     */
    private String separator;

    /**
     * 默认 java.lang.Long
     */
    private Class elementType;

    /**
     * 默认 ArrayList
     */
    private Class collectionType;

    @Override
    public void setParameterValues(Properties parameters) {
        String separator = (String)parameters.get("separator");
        if(!StringUtils.isEmpty(separator)) {
            this.separator = separator;
        } else {
            this.separator = ",";
        }
    }
}
```

```
String collectionType = (String)parameters.get("collectionType");
if(!StringUtils.isEmpty(collectionType)) {
    try {
        this.collectionType = Class.forName(collectionType);
    } catch (ClassNotFoundException e) {
        throw new HibernateException(e);
    }
} else {
    this.collectionType = java.util.ArrayList.class;
}

String elementType = (String)parameters.get("elementType");
if(!StringUtils.isEmpty(elementType)) {
    try {
        this.elementType = Class.forName(elementType);
    } catch (ClassNotFoundException e) {
        throw new HibernateException(e);
    }
} else {
    this.elementType = Long.TYPE;
}
}

@Override
public int[] sqlTypes() {
    return new int[] {Types.VARCHAR};
}

@Override
public Class returnedClass() {
    return collectionType;
}

@Override
public boolean equals(Object o, Object o1) throws HibernateException {
    if (o == o1) {
```



```
        return true;
    }
    if (o == null || o1 == null) {
        return false;
    }

    return o.equals(o1);
}

@Override
public int hashCode(Object o) throws HibernateException {
    return o.hashCode();
}

/**
 * 从JDBC ResultSet读取数据,将其转换为自定义类型后返回
 * (此方法要求对可能出现null值进行处理)
 * names中包含了当前自定义类型的映射字段名称
 * @param names
 * @param owner
 * @return
 * @throws HibernateException
 * @throws java.sql.SQLException
 */
@Override
public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object
    String valueStr = rs.getString(names[0]);
    if(StringUtils.isEmpty(valueStr)) {
        return null;
    }

    String[] values = StringUtils.split(valueStr, separator);

    Collection result = newCollection();

    for(String value : values) {
```

```
        result.add(ConvertUtils.convert(value, elementType));
    }
    return result;
}

private Collection newCollection() {
    try {
        return (Collection)collectionType.newInstance();
    } catch (Exception e) {
        throw new HibernateException(e);
    }
}

/**
 * 本方法将在Hibernate进行数据保存时被调用
 * 我们可以通过PreparedStatement将自定义数据写入到对应的数据库表字段
 */
@Override
public void nullSafeSet(PreparedStatement st, Object value, int index, SessionImplementor s)
    throws SQLException {
    String valueStr;
    if(value == null) {
        valueStr = "";
    } else {
        valueStr = StringUtils.join((Collection)value, separator);
    }
    st.setString(index, valueStr);
}

/**
 * 提供自定义类型的完全复制方法
 * 本方法将用构造返回对象
 * 当nullSafeGet方法调用之后，我们获得了自定义数据对象，在向用户返回自定义数据之前，
 * deepCopy方法将被调用，它将根据自定义数据对象构造一个完全拷贝，并将此拷贝返回给用户
 * 此时我们就得到了自定义数据对象的两个版本，第一个是从数据库读出的原始版本，其二是我们通过
```

```
* deepCopy方法构造的复制版本，原始的版本将有Hibernate维护，复制版由用户使用。原始版本用作
* 稍后的脏数据检查依据；Hibernate将在脏数据检查过程中将两个版本的数据进行对比（通过调用
* equals方法），如果数据发生了变化（equals方法返回false），则执行对应的持久化操作
*
* @param o
* @return
* @throws HibernateException
*/
@Override
public Object deepCopy(Object o) throws HibernateException {
    if(o == null) return null;
    Collection copyCollection = newCollection();
    copyCollection.addAll((Collection)o);
    return copyCollection;
}

/**
 * 本类型实例是否可变
 * @return
 */
@Override
public boolean isMutable() {
    return true;
}

/* 序列化 */
@Override
public Serializable disassemble(Object value) throws HibernateException {
    return ((Serializable)value);
}

/* 反序列化 */
@Override
public Object assemble(Serializable cached, Object owner) throws HibernateException {
    return cached;
}
```

```
@Override
public Object replace(Object original, Object target, Object owner) throws HibernateException {
    return original;
}

}
```

1、setParameterValues 作用是参数化 集合类型 和分隔符 不写死了

2、deepCopy??????? ???????? session??????????

使用：

```
/**
 * 此处没有使用关联 是为了提高性能（后续会挨着查询资源和权限列表，因为有缓存，数据量也不是很大 所以性能
 * <p>User: Zhang Kaitao
 * <p>Date: 13-4-5 下午2:04
 * <p>Version: 1.0
 */

@TypeDef(
    name = "SetToStringUserType",
    typeClass = CollectionToStringUserType.class,
    parameters = {
        @Parameter(name = "separator", value = ","),
        @Parameter(name = "collectionType", value = "java.util.HashSet"),
        @Parameter(name = "elementType", value = "java.lang.Long")
    }
)

@Entity
```

```
@Table(name = "sys_role_resource_permission")
public class RoleResourcePermission extends BaseEntity<Long> {

    /**
     * 角色id
     */
    @ManyToOne(optional = true, fetch = FetchType.EAGER)
    @Fetch(FetchMode.SELECT)
    private Role role;

    /**
     * 资源id
     */
    @Column(name = "resource_id")
    private Long resourceId;

    /**
     * 权限id列表
     * 数据库通过字符串存储 逗号分隔
     */
    @Column(name = "permission_ids")
    @Type(type = "SetToStringUserType")
    private Set<Long> permissionIds;

    public RoleResourcePermission() {
    }
}
```

@TypeDef(

name = "SetToStringUserType",

typeClass = CollectionToStringUserType.class,

parameters = {

@Parameter(name = "separator", value = ","),

@Parameter(name = "collectionType", value = "java.util.HashSet"),

```
        @Parameter(name = "elementType", value = "java.lang.Long")

    }

)
```

定义类型并指定参数化的集合类型、元素类型和分隔符。

[github代码](#)