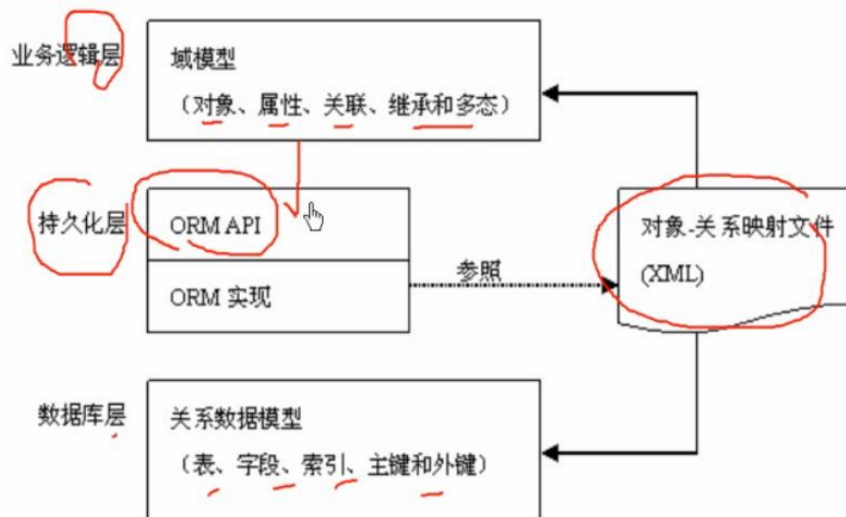


Hibernate 视频教程

原载：51CTO - 疯狂软件

http://edu.51cto.com/course/course_id-431.html

ORM



Hibernate 与 Jdbc 代码对比

```
public void save(Session sess, Message m) {  
    sess.save(m);  
}
```

Hibernate 实现

```
public void save(Connection conn, Message m) {  
    PreparedStatement ps = null;  
    String sql = "insert into message values (?, ?)";  
  
    try {  
        ps = conn.prepareStatement(sql);  
        ps.setString(1, m.getTitle());  
        ps.setString(2, m.getContent());  
        ps.execute();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        if (ps != null) {  
            try {  
                ps.close();  
            } catch (SQLException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

JDBC 实现

Hibernate介绍

Hibernate是目前最流行的ORM框架之一，它是一个面向Java环境的对象/关系数据库映射工具。

Hibernate也是一个轻量级的O/R Mapping框架，是目前最流行的持久层解决方案，较之另一个持久层框架MyiBatis，Hibernate更具有面向对象的特征；较之传统的EJB的持久层解决方案，Hibernate则采用低侵入式的设计，即完全采用普通的Java对象(POJO,VO)，而不必继承Hibernate的某个超类或实现Hibernate的某个接口。
Hibernate是面向对象的程序设计语言和关系数据库之间的桥梁，真正实现了开发者采用面向对象的方式来操作关系数据库。

Hibernate的作用

1. 让我们可以使用面向对象的方式更方便进行持久化相关操作的开发。
2. 提高开发效率。

Hibernate不一定可以提高程序的性能！

Hibernate不一定可以提高可扩展性、可维护性！

Hibernate是ORM规范的实现框架，

所有ORM框架的作用：负责把面向对象的持久化操作，转换为数据库标准SQL语句去执行。

ORM规范映射思想：

- 一个表 映射 成一个类。
- 一行记录（一条数据）映射 成一个 对象。
- 一列（一个字段）映射 成 对象 的属性。

Hibernate 不适合关系特别复杂的数据库，也不如 JDBC 快

3-配置hibernate.cfg.xml文件

负责管理与数据库连接的相关信息

配置文件有两种方式：

- 1: hibernate.properties
- 2: hibernate.cfg.xml — 可以在配置文件中加载持久化类。

```
Configuration cfg = new Configuration();
cfg.addFile("src/orq/fkjava/bean/Person.hbm.xml");

Configuration cfg = new Configuration().configure();
```

FK 疯狂Java——技术沉淀最厚的高级软件培训专家

编写映射(mapping)文件

- 几乎所有的ORM工具都需要这样一个数据库字段与JavaBean属性匹配的映射文件。在hibernate中习惯的命名为“*.hbm.xml”文件，如代码所示。一般情况下建议映射文件与其对应的class文件在同一级目录中，这样也方便查

Person类
-id
-name
-age
+toString()

Person.hbm.xml

PERSON表
-ID
-NAME
-AGE

疯狂源自梦想 技术成就辉煌

疯狂软件 www.fkjava.org

- * 需要注意:
- * 1: 通常这个类需要有一个 id , 一般建议使用封装类型
- * 2: 这个类不能是final修饰的
- * 3: 需要给这个类提供一个无参的构造器
- * 4: 需要给所有的属性提供getting/setting方法
- * 5: 如果有涉及集合数据的操作,集合类型要使用接口类型 List ArrayList
- *

Configuration

概述:

Configuration 类负责管理Hibernate 的配置信息。它包括如下内容:

- Hibernate运行的底层信息: 数据库的URL、用户名、密码、JDBC驱动类, 数据库Dialect,数据库连接池等。
- Hibernate映射文件 (*.cfg.xml)。
- 一般存放在classpath中
- **Hibernate配置**的两种方法:
- 属性文件 (hibernate.properties)

调用代码: Configuration cfg = new Configuration();
cfg.addFile("src/org/fkjava/bean/Person.hbm.xml");

- Xml文件 (hibernate.cfg.xml)

调用代码: Configuration cfg = new Configuration().configure();

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 2014-12-17 12:14:51 by Hibernate Tools 3.4.0.CR1 -->
<hibernate-mapping package="org.fkjava.pojo">

    <class name="Person" table="t_PERSON" dynamic-update="true" >
        <id name="ID">
            <column name="ID" />
            <generator class="hilo" />
        </id>

        <property name="name" >
            <column name="NAME" />
        </property>
        <property name="password" >
            <column name="PASSWORD" />
        </property>
        <property name="birthday" >
            <column name="BIRTHDAY" />
        </property>

        <union-subclass name="Student" table="t_student">
            <property name="number" column="NUMBER"></property>
            <property name="score" column="SCORE"></property>
        </union-subclass>

    </class>
</hibernate-mapping>
```


SessionFactory

概述:

应用程序从SessionFactory（会话工厂）里获得Session(会话)实例。它在**多个线程间进行共享**。通常情况下，整个应用只有**唯一**的一个会话工厂——例如在应用初始化时被创建。然而，如果你使用Hibernate访问多个数据库，你需要对每一个数据库使用一个会话工厂。会话工厂缓存了生成的SQL语句和Hibernate在运行时使用的映射元数据。

• 调用代码:

```
SessionFactory sessionFactory = cfg.buildSessionFactory();
```

Session(会话)

概述:

Session非线程安全的，它代表与数据库之间的一次操作，它的概念介于Connection和Transaction之间。

- Session也称为**持久化管理器**，因为它是与持久化相关的操作接口。
- Session通过**SessionFactory**打开，在所有的工作完成后，需要关闭。
- 它与Web层的HttpSession没有任何关系。

• 调用代码

```
Session session = sessionFactory.openSession();
```

Persist()暂时不保证一定执行，返回值为空

Save 保证一定立刻执行，long conversation 表现不好，返回标示符

```
import org.hibernate.*;
import org.hibernate.cfg.*;
```

Hibernate测试类

```
public class TestHibernate {
    public static void main(String[] args) {
        // Configuration 负责管理 Hibernate 配置信息
        Configuration config = new Configuration().configure();
        // 通过 config 建立 SessionFactory
        // 通过 SessionFactory 获得 Session
        ServiceRegistry sr = new
        ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();

        SessionFactory factory = config.buildSessionFactory(sr);
        // 将持久化的对象
        Person person = new Person();
        person.setName("hanfeili");
        person.setAge(new Integer(30));
        // 开启 Session, 相当于开启 JDBC 的 Connection
        Session session = sessionFactory.openSession();
        // Transaction 表示一组对 DB 的事务
        Transaction tx = session.beginTransaction();
        // 将对象映像至数据库表格中储存
        session.save(person); // session.persist(person);
        tx.commit();
    }
}
```

疯狂软件 技术成就辉煌

疯狂软件 www.fkjava.org

```
private static void testUpdate3() {
    Configuration config = new Configuration().configure();
    //hibernate4 推荐实现服务注册方式进行 SessionFactory 获取
    ServiceRegistry sr = new ServiceRegistryBuilder()
        .applySettings(config.getProperties()).buildServiceRegistry();

    SessionFactory factory = config.buildSessionFactory(sr);
    Session session = factory.openSession();
    Transaction tx = session.beginTransaction();
    //持久化对象在更新的时候可以不需要显示的调用 update() 方法
    //持久化状态对象会在 session 关闭的时候如果内存数据和表数据不一致将自动 同步 到数据库表。
    Person p = (Person) session.get(Person.class, 5);
    p.setName("tom9999");
    //更新
    session.update(p);

    tx.commit();

    session.close();
}
```

Hibernate 三种状态的区分, 以及

save, update, saveOrUpdate, merge 等的使用

分类: [Hibernate](#) 2013-04-14 15:33 675 人阅读 [评论\(0\)](#) [收藏](#) [举报](#)

[hibernatesaveupdatesaveOrUpdatemerge](#)

Hibernate 的对象有 3 种状态，分别为：瞬时态(Transient)、持久态(Persistent)、脱管态(Detached)。处于持久态的对象也称为 PO(Persistence Object)，瞬时对象和脱管对象也称为 VO (Value Object)。

瞬时态

由 new 命令开辟内存空间的 java 对象，

eg. `Person person = new Person("xxx", "xx");`

如果没有变量对该对象进行引用，它将被 java 虚拟机回收。

瞬时对象在内存孤立存在，它是携带信息的载体，不和数据库的数据有任何关联关系，在 Hibernate 中，可通过 session 的 `save()`或 `saveOrUpdate()`方法将瞬时对象与数据库相关联，并将数据对应的插入数据库中，此时该瞬时对象转变成持久化对象。

持久态

处于该状态的对象在数据库中具有对应的记录，并拥有一个持久化标识。如果是用 hibernate 的 `delete()`方法，对应的持久对象就变成瞬时对象，因数据库中的对应数据已被删除，该对象不再与数据库的记录关联。

当一个 session 执行 `close()`或 `clear()`、`evict()`之后，持久对象变成脱管对象，此时持久对象会变成脱管对象，此时该对象虽然具有数据库识别值，但它已不在 Hibernate 持久层的管理之下。

持久对象具有如下特点：

1. 和 session 实例关联；
2. 在数据库中有与之关联的记录。

脱管态

当与某持久对象关联的 session 被关闭后，该持久对象转变为脱管对象。当脱管对象被重新关联到 session 上时，并再次转变成持久对象。

脱管对象拥有数据库的识别值，可通过 `update()`、`saveOrUpdate()`等方法，转变成持久对象。

脱管对象具有如下特点：

1. 本质上与瞬时对象相同，在没有任何变量引用它时，JVM 会在适当的时候将它回收；
2. 比瞬时对象多了一个数据库记录标识值。

hibernate 的各种保存方式的区(`save,persist,update,saveOrUpdte,merge,flush,lock`)及 对象的三种状态

hibernate 的保存

hibernate 对于对象的保存提供了太多的方法，他们之间有很多不同，这里细说一下，以便区别。

一、预备知识

对于 hibernate，它的对象有三种状态，transient、persistent、detached

下边是常见的翻译办法：

transient：瞬态或者自由态

(new DeptPo(1,"行政部",20,"行政相关")，该 po 的实例和 session 没有关联，该 po 的实例处于 transient)

persistent：持久化状态

(和数据库中记录想影射的 Po 实例，它的状态是 persistent，通过 get 和 load 等得到的对象都是 persistent)

detached：脱管状态或者游离态

(1)当通过 get 或 load 方法得到的 po 对象它们都处于 persistent,但如果执行 delete(po)时(但不能执行事务),该 po 状态就处于 detached, (表示和 session 脱离关联),因 delete 而变成游离态可以通过 save 或 saveOrUpdate()变成持久态

(2)当把 session 关闭时，session 缓存中的 persistent 的 po 对象也变成 detached

因关闭 session 而变成游离态的可以通过 lock、save、update 变成持久态

持久态实例可以通过调用 delete()变成脱管状态。

通过 get()或 load()方法得到的实例都是持久化状态的。

脱管状态的实例可以通过调用 lock()或者 replicate()进行持久化。

save()和 persist()将会引发 SQL 的 INSERT，delete()会引发 SQLDELETE，

而 update()或 merge()会引发 SQL UPDATE。对持久化（persistent）实例的修改在刷新提交的时候会被检测到，它也会引起 SQL UPDATE。

saveOrUpdate()或者 replicate()会引发 SQLINSERT 或者 UPDATE

二、save 和 update 区别

把这一对放在第一位的原因是因为这一对是最常用的。

save 的作用是把一个新的对象保存

update 是把一个脱管状态的对象或自由态对象（一定要和一个记录对应）更新到数据库

三、update 和 saveOrUpdate 区别

这个是比较好理解的，顾名思义，saveOrUpdate 基本上就是合成了 save 和 update,而

update 只是 update;引用 hibernate reference 中的一段话来解释他们的使用场合和区别

通常下面的场景会使用 update()或 saveOrUpdate():

程序在第一个 session 中加载对象,接着把 session 关闭

该对象被传递到表现层

对象发生了一些改动

该对象被返回到业务逻辑层最终到持久层

程序创建第二 session 调用第二个 session 的 update()方法持久这些改动

`saveOrUpdate(po)`做下面的事:

如果该 `po` 对象已经在本 `session` 中持久化了, 在本 `session` 中执行 `saveOrUpdate` 不做任何事

如果 `saveOrUpdate(新 po)`与另一个与本 `session` 关联的 `po` 对象拥有相同的持久化标识(`identifier`), 抛出一个异常

`org.hibernate.NonUniqueObjectException: a different object with the same identifier value was already associated with the session: [org.itfuture.www.po.Xtyhb#5]`

`saveOrUpdate` 如果对象没有持久化标识(`identifier`)属性, 对其调用 `save()`, 否则 `update()` 这个对象

四、`persist` 和 `save` 区别

这个是最迷离的一对, 表面上看起来使用哪个都行, 在 `hibernate reference` 文档中也没有明确的区分他们.

这里给出一个明确的区分。(可以跟进 `src` 看一下, 虽然实现步骤类似, 但是还是有细微的差别)

主要内容区别:

1, `persist` 把一个瞬态的实例持久化, 但是并不"保证"标识符(`identifier` 主键对应的属性)被立刻填入到持久化实例中, 标识符的填入可能被推迟到 `flush` 的时候。

2, `save`, 把一个瞬态的实例持久化标识符, 及时的产生, 它要返回标识符, 所以它会立即执行 `Sql insert`

五、`saveOrUpdate`, `merge` 和 `update` 区别

比较 `update` 和 `merge`

`update` 的作用上边说了, 这里说一下 `merge` 的

如果 `session` 中存在相同持久化标识(`identifier`)的实例, 用用户给出的对象覆盖 `session` 已有的持久实例

(1)当我们使用 `update` 的时候, 执行完成后, 会抛出异常

(2)但当我们使用 `merge` 的时候, 把处理自由态的 `po` 对象 `A` 的属性 `copy` 到 `session` 当中处于持久态的 `po` 的属性中, 执行完成后原来是持久状态还是持久态, 而我们提供的 `A` 还是自由态

六、`flush` 和 `update` 区别

这两个的区别好理解

`update` 操作的是在自由态或脱管状态(因 `session` 的关闭而处于脱管状态)的对象

//updateSQL

而 `flush` 是操作的在持久状态的对象。

默认情况下, 一个持久状态的对象改动(包含 `set` 容器)是不需要 `update` 的, 只要你更改了对象的值, 等待 `hibernate flush` 就自动更新或保存到数据库了。`hibernate flush` 发生

在以下几种情况中：

1， 调用某些查询的和手动 flush(),session 的关闭、SessionFactory 关闭结合 get()一个对象，把对象的属性进行改变,把资源关闭。

2， transaction commit 的时候（包含了 flush）

七、lock 和 update 区别

update 是把一个已经更改过的脱管状态的对象变成持久状态

lock 是把一个没有更改过的脱管状态的对象变成持久状态(针对的是因 Session 的关闭而处于脱管状态的 po 对象(2)，不能针对因 delete 而处于脱管状态的 po 对象)

对应更改一个记录的内容，两个的操作不同：

update 的操作步骤是：

(1)属性改动后的脱管的对象的修改->调用 update

lock 的操作步骤是：

(2)调用 lock 把未修改的对象从脱管状态变成持久状态-->更改持久状态的对象的内容-->等待 flush 或者手动 flush

八、clear 和 evict 的区别

clear 完整的清除 session 缓存

evict(obj)把某个持久化对象从 session 的缓存中清空。

```
ServiceRegistry sr = new ServiceRegistryBuilder()
    .applySettings(config.getProperties()).buildServiceRegistry();

SessionFactory factory = config.buildSessionFactory(sr);
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

Person p = new Person();
p.setId(60);
p.setName("B");
//新增或更新，根据id是否存在，id存在就是更新，不存在就是新增
session.saveOrUpdate(p);

tx.commit();

session.close();
}
```

如果给定的 ID 是不存在在数据库里的，那么会报错，因为有 ID 就不会新增，但是也不会 Update。但是如果执行 save()就能成功，只是真正存入数据库的 ID 由 config XML 文件决定

Note that Load() will throw an unrecoverable exception if there is no matching database row. If the class is mapped with a proxy, Load() returns an object that is an uninitialized proxy and does not actually hit the database until you invoke a method of the object. This behaviour is very useful if you wish to create an association to an object without actually loading it from the database.

If you are not certain that a matching row exists, you should use the Get() method, which hits the database immediately and returns null if there is no matching row.

```

Transaction tx = session.beginTransaction();

Person p = (Person)session.load(Person.class, 6);

session.delete(p);

tx.commit();

```

```

Transaction tx = session.beginTransaction();

Person p = new Person();
p.setId(5);
session.delete(p);

tx.commit();
session.close();

```

删除临时化和持久化状态（没有则抛出异常）

```

@Before
public void setUp() throws Exception {
    Configuration configuration = new Configuration();

    configuration.configure();

    ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().
        applySettings(configuration.getProperties()).build();

    factory = configuration.buildSessionFactory(serviceRegistry);
    session = factory.openSession();
}

@After
public void tearDown() throws Exception {
    if (session.isOpen()) {
        session.close();
    }
}

@Test
public void test() {
    transaction = session.beginTransaction();

    Person person = new Person("admin", 123456, new Date());
    person.setID(new Integer(70));
    session.save(person);
    transaction.commit();
}

```

JUnit 测试

```
<hibernate-mapping default-access="field">
```

这里 default 是 property, 通过 getter 和 setter 访问。也可以使用 field, 不通过 getter 和 setter 访问

default-access="field" 对象属性状态操作方式设置
property 表示属性需要通过getting/setting方法操作（默认）
field表示可以不需要getting/setting方法就可以通过反射的方式操作属性

package 指定本持久化配置文件中的class的包名
default-lazy 指定本持久化配置文件中的class延迟加载策略
default-cascade 指定本持久化配置文件中的class级联操作策略

只有 load 或者 get 的持久对象才能使用 dynamic-update 或者 dynamic-insert 属性

```
<class
name="ClassName"                (1)
table="tableName"               (2)
mutable="true|false"           (3)
dynamic-update="true|false"     (4)
dynamic-insert="true|false"     (5)
select-before-update="true|false" (6)
where="arbitrary sql where condition" (7)
formula="(select count(*) from table t_person)" (8) />
```

mutable 表明该类是否允许更新。如果将它设为false，则应用程序不能对此类对应的数据进行修改。可以insert，delete，不能update操作

dynamic-update（动态更新）
select-before-update 指定Hibernate除非确定对象的确被修改了，不会执行SQL UPDATE操作。在特定场合（实际上，只会发生在一个临时对象关联到一个新的session中去，执行update()的时候），这说明Hibernate会在UPDATE之前执行一次额外的SQL SELECT操作，来决定是否应该进行UPDATE。

where（可选） 指定一个附加的SQLWHERE条件，在抓取这个类的对象时会一直增加这个条件。

formula 给类中的派生列赋值，通过查询语句将查询的结果赋值一个派生列。

Formula 只能用在 property 上，而且类里头要有 getter 和 setter

```
<property name="count" formula="1" /></property>
```

1. increment:

用途: 适用于int, short, long类型的主键, 每次主键自增1

缺点: 不能在集群情况下使用, 并发操作数据库时, 多个实例各自维护自己的主键状态, 会发生冲突

2. identity:

用途: 适用于内部支持标识字段的数据库(mysql, postgres, mssql)

3. sequence:

用途: 适用于内部支持序列的数据库(db2, oracle, postgres)

用法: 必须在数据库先创建一个序列, 并且在hibernate配置文件中对param进行配置

4. hilo:

用途: 通过hi/lo算法来生成主键

缺点: 当使用数据库连接池时, 不可以使用, 因为检索hi值的sql语句必须在一个独立的事务中完成, 因此生成器必须获得新的connection

5. native:

用途: 根据使用的数据库自行判断使用identity, sequence, hi/lo

Oracle 不能自增主键

序列配置实例

```
<id name="id">
  <generator class="sequence">    <!--指定id生成策略使用序列方式-->
    <param name="sequence">ff_seq</param>    <!--指定oracle数据库sequence名称-->
  </generator>
</id>
---
```

--序列 (oracle没有id自动增长功能, 所以需要自己创建序列来实现) ---

```
CREATE SEQUENCE ff_seq
  INCREMENT BY 1 --每一次增长多少
  START WITH 1 --第一次从什么位置开始
  maxvalue 999999999 --最大值是多少
```

--删除序列

```
drop sequence ff_seq;
```

--nextval 获得下一个值

```
select ff_seq.NEXTVAL from dual;
```

--currval 获得当前值

```
select ff_seq.CURRVAL from dual;
```

--序列使用实例

```
create table t_seq(
  id int primary key,
  name varchar2(50) not null
```


Property 配置实例

```
<property
name="propertyName"           (1)
column="column_name"         (2)
type="typename"              (3)
length=10                    (4)
precision                     (5)
scale                        (6)
not-null="true | false"      (7)
update="true | false"        (8)
insert="true | false"        (9)
unique="true | false"        (10)
unique-key="true | false"    (11)
lazy=true/false              (12)
index                        (13)
not-null (14) />
```

(5) 指定数值的有效位数

(6) 指定小数的位数

(8-9) update, insert :表明在用于UPDATE 和/或 INSERT的SQL语句中是否包含这个字段。

(10) 唯一约束

(11) 唯一约束名

(12) 指定是否延迟加载

(13)用于为该列建索引

基本类型映射表

Hibernate	Java类型	标准SQL类型
integer/int	int/java.lang.Integer-integer	
long	long/java.lang.Long	bigint
short	short/java.lang.Short	smallint
byte	byte/java.lang.Byte	tinyint
float	float/java.lang.Float	float
double	double/java.lang.Double	double
big_decimal	java.math.BigDecimal	numeric
character	Char/Character/String	char(1)
string	java.lang.String	varchar
boolean	boolean/java.lang.Boolean	bit
yes_no	boolean/java.lang.Boolean	char(1)('Y' /' N')
true_false	boolean/java.lang.Boolean	char(1)('T' /' F')

大对象配置表

映射类型	Java类型	标准SQL类型	描述
binary	byte[]	varbinary/blob	存放二进制数
text	java.lang.String	clob	字符串大对象
serializable	实现java.io.Serializable接口的类	varbinary/blob	
clob	java.sql.Clob	clob	字符串大对象
blob	java.sql.Blob	blob	二进制大对象

自定义类型实现。

可以实现org.hibernate.UserType或org.hibernate.CompositeUserType中的任一个，并且使用类型的Java全限定类名来定义属性。

Session一级缓存

一级缓存生命周期很短与session生命周期一致，所以一级缓存也叫session级缓存或事务级缓存。位于缓存中的对象处于持久化状态，它和表中的相关记录对应，Session能够在某些时间点，按照缓存中持久化对象的属性变化来同步数据库中表的记录，这一过程称为清理缓存。

(1) 一级缓存实现原理。Session缓存是由它的实现类SessionImpl中定义的一些集合属性构成的，原理是保证有一个引用在关联着某个持久化对象，保持它的生命周期不会结束。

(2) Session缓存的作用。减少数据库访问，从内存中取数据比数据库中要快的多。缓存中的数据与数据库中的同步：缓存会把改变的sql语句合并，减少访问次数。缓存中的对象存在循环关联时，session会保证不出现访问对象图的死循环。

session缓存对象的存储位置在：

session->persistenceContext->entityEntries->map

缓存对象的状态

```
{Person [id=3, name=tom2, age=18, birthday=2013-08-10
14:45:16.0]=EntityEntry[org.fkjava.pojo.Person#3] (MANAGED)}
```

只看 Session 关闭时候对象的差距。调用多次 session.get 得到的对象是相同的
Session.clear(); 会再查

Session 的 save 和 evict 方法

问：

先创建一个 Student,然后调用 session.save 方法，然后再调用 evict 方法把 Student 对象清除出缓存，再提交事务，

可是会报错：Exception in thread "main" org.hibernate.AssertionFailure: possible nonthreadsafe access to session

但是如果我用的不是 evict 方法，而是 clear 方法用来清除缓存的话，程序没有错。

答：

session.evict(obj)，会把指定的缓冲对象进行清除

session.clear()，把缓冲区内的全部对象清除，但不包括操作中的对象

所以，hibernate 执行的顺序如下，

(1)生成一个事务的对象，并标记当前的 Session 处于事务状态（注：此时并未启动数据库级事务）。

(2)应用使用 s.save 保存对象，这个时候 Session 将这个对象放入 entityEntries，用来标记对象已经和当前的会话建立了关联，由于应用对对象做了保存的操作，Session 还要在 insertions 中登记应用的这个插入行为（行为包括：对象引用、对象 id、Session、持久化处理类）。

(3)s.evict 将对象从 s 会话中拆离，这时 s 会从 entityEntries 中将这个对象移出。

(4)事务提交，需要将所有缓存 flush 入数据库，Session 启动一个事务，并按照 insert,update,……,

delete 的顺序提交所有之前登记的操作（注意：所有 insert 执行完毕后会才会执行 update，这里的特殊处理也可能会将你的程序搞得一团糟，如需要控制操作的执行顺序，要善于使用 flush），现在对象不在 entityEntries 中，但在执行 insert 的行为时只需要访问 insertions 就足够了，所以此时不会有任何的异常。异常出现在插入后通知 Session 该对象已经插入完毕这个步骤上，这个步骤中需要将 entityEntries 中对象的 existsInDatabase 标志置为 true，由于对象并不存在于 entityEntries 中，此时 Hibernate 就认为 insertions 和 entityEntries 可能因为线程安全的问题产生了不同步（也不知道 Hibernate 的开发者是否考虑到例子中的处理方式，如果没有的话，这也许算是一个 bug 吧），于是一个 net.sf.hibernate.AssertionFailure 就被抛出，程序终止

一般错误的认为 s.save 会立即的执行，而将对象过早的与 Session 拆离，造成了 Session 的 insertions 和 entityEntries 中内容的不同步。所以我们在做此类操作时一定要清楚 Hibernate 什么时候会将数据 flush 入数据库，在未 flush 之前不要将已进行操作的对象从 Session 上拆离。

解决办法是在 save 之后，添加 session.flush。

批量操作提高程序效率：session.flush(); session.clear(); flush 并不清空缓存，只有 clear 清空

组件类的配置

```
<!-- 组件类的映射配置
    component指定需要映射的组件类
    name指定Person中组件属性的变量名称
    property 指定组件类中什么属性参与映射
-->
<component name="phone">
    <property name="companyPhone" column="t_company_phone"/>
    <property name="homePhone"/>
</component>
```

成员不是基本函数

没有用 class 定义的都是组件，定义的都是有自己表的。无论静态还是动态，都会存在原来的表里面

```
<!-- 动态组件类的映射配置
    name="attribute" 对应持久化类中集合的变量名称
    property Map集合中key映射配置
    name 对应map集合的key
    column指定map中的key对应的value存储到什么列中
    type 指定列的类型
-->
<dynamic-component name="attribute">
    <property name="key1" column="t_key1" type="string"/>
    <property name="key2" column="t_key2" type="integer"/>
    <property name="key3" column="t_key3" type="boolean"/>
    <property name="key4" column="t_key4" type="double"/>
</dynamic-component>
```

Map，list 和 set 基础配置

Map 等在加入基本类型时都会包装

```

<array name="arrays">
  <key column="id"></key>
  <list-index column="indices"></list-index>
  <element column="value" type="string"></element>
</array>

<list name="lists">
  <key column="id"></key>
  <list-index column="indices"></list-index>
  <element column="value" type="string"></element>
</list>

<map name="maps">
  <key column="id"></key>
  <map-key column="map_key" type="string"></map-key>
  <element column="value" type="string"></element>
</map>

<set name="sets">
  <key column="id"></key>
  <element column="value" type="string"></element>
</set>

```

Map 的 key 也一定要设置类型

Bag 配置

bag 可以重复存放数据

但是修改和删除的时候因为没有id值，所以不知道具体是哪一条，所以会全部删除，重新插入数据

```

<bag name="items" table="item">
  <key column="id" />
  <element type="java.lang.String" column="name" />
</bag>

```

idbag （是对bag的一个升级）

数据带 id

```

<idbag name="items" table="item">
  <collection-id column="cid" type="java.lang.String">
    <generator class="uuid.hex" />
  </collection-id>
  <key column="id" />
  <element column="name" type="java.lang.String" />
</idbag>

```

Idbag 需要算法来算高低值，不常用

关联配置

一对一 外键双向关联配置

双向关联

双向关联关系：

就是 关联关系的双方可以互相访问对方



单向关联关系

就是只能一方访问一方，不能互相访问

双向关联关系砍掉一边就是单向关联关系。

所以我们重点介绍双向关联关系，单向关联关系可以参考API

```
<!--
    one-to-one 表示一个一对一的配置
    name指定Person类中的关联关系的address属性
    cascade 表示级联操作
    级联是对象之间的关联操作，只影响添加、删除和更新，不影响 查询。
    all: 所有情况下都进行关联操作 (save, update, delete)
    none: 所有情况下都不进行关联操作。(默认值)
    save-update:在执行 (save, update, saveOrUpdate) 进行关联操作。
    delete: 在执行delete时进行关联操作。
-->
```

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

但是 cascade 能否成功取决于数据库的设置，而非 Hibernate 的设置

```
<!-- 基于外键的的一对一的配置在从表的一端使用 many-to-one
    name Address类中关联关系Person类的引用变量名
    column 指定外键列名,如果不指定person_id
    unique 唯一约束,因为使用的一对一外键,使用外键列必须唯一
    not-null=true 指定列不能为空
-->
<many-to-one name="person" column="p_id" unique="true" not-null="true"/>
```

Many-to-one + unique = one - to - one slave 注意 name 和 column 必须在从表里指定

配置一个持久化类的步骤: 1. 写出类 (有 ID, 有 getter 和 setter) 2. 写出 xml 文件。3. 把 xml 文件导入 hibernate.cfg.xml 里

如果是双向的连接关系，应该两边都有一个指向对方的引用

```

person.setAddresses(addresses);
addresses.setPerson(person);
session.save(person);
session.save(addresses);

```

数据保存的时候，先存储主表，再存储从表，否则会执行 update/失败

```

/**
 * 通过主表获得从表数据
 * 通过一个left join 在一条语句中进行查询
 */
@Test
public void testOne2One2Get() {
    Person p = (Person) session.get(Person.class, 1);
    System.out.println(p);
    System.out.println("-----");
    System.out.println(p.getAddress());
}

/**
 * 通过从表获得主表数据
 * 在使用主表数据的时候才通过一个left join 在一条语句中进行查询（延迟查询）
 */
@Test
public void testOne2One2Get2() {
    Address address = (Address) session.get(Address.class, 1);
    System.out.println(address);
    System.out.println("-----");
    System.out.println(address.getPerson());
}

```

一对一 主键双向关联配置

```

<!-- 在主键一对一的从表中从表的id主键不能自己产生，必须与主表保持一致，所以通过主表获取id就好了 -->
<id name="id">
    <generator class="foreign">
        <!-- 指定从表的id 来源 -->
        <param name="property">person</param>
    </generator>
</id>

<property name="descs" />
<property name="code" />
<!-- 基于主键的一对一的配置在从表的一端使用 one-to-one
      name Address类中关联关系Person类的引用变量名
      constrained 启用约束控制
-->
<one-to-one name="person" constrained="true"/>

```

一对多 外键双向关联配置

```
private Set<Addresses> addressesSet = new HashSet<>();

<!-- 一对多 双向关联
    在一个人有多个地址（人有家庭地址，工作地址）
    在一端（Person）所以set 表示有关联多的一端（Address）
    inverse="true" 将关系控制交给多的一端维护，效率更高
-->
<set name="addresses" cascade="all" inverse="true">
    <!-- 多的一端使用key指定的column跟一的一端建立关联关系
        所以key的column 必须 与在多的那一端的配置中使用的
        <many-to-one ... column="person_id"/>的column一致
    -->
    <key column="person_id"/>
    <!-- 告诉系统多的一端是什么类型的数据 -->
    <one-to-many class="Address"/>
</set>
</class>

<!-- 因为现在使用的一对多 所以将之前的一对一中配置的 unique="true" 删除就可以 -->
<many-to-one name="person" column="person_id" /> 注意对比 一对一外键关联 的配置

<set name="addressesSet" cascade="all" inverse="true">
    <key column="person_id"/></key>
    <one-to-many class="org.fkjava.pojo.Addresses"/>
</set>
```

必须相同

Getter 和 Setter 很重要！类名要写全，一定要删 unique

```
/*Addresses addresses1 = new Addresses("111122222", 111222);
Addresses addresses2 = new Addresses("111122223", 111222);
Addresses addresses3 = new Addresses("111122224", 111222);

Set<Addresses> addresses = new HashSet<>();

addresses.add(addresses1);
addresses.add(addresses2);
addresses.add(addresses3);

Person person = new Person("admin", 123456, new Date());
person.setPhones(new Phones("sssss", "ssssss"));

addresses1.setPerson(person);
addresses2.setPerson(person);
addresses3.setPerson(person);

person.setAddressesSet(addresses);

session.save(person);*/
```

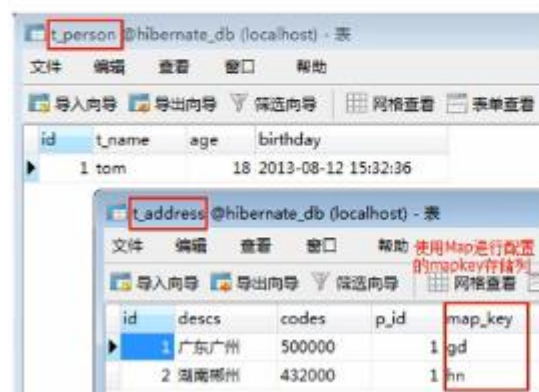
用 list 来配置：注意 inverse 不能为 true

一对多，一的一边需要配置 Set，然后用 inverse=true 将控制权交给多的一方

```
<!-- 如果希望多端的数据有顺序可以使用list进行配置
      inverse写true a_index将不会有数据
      因为inverse是关系的控制，这里address对应数据的顺序只要Person知道
      ，所以不能设置为true了
-->
<list name="addresss" inverse="false" cascade="all">
  <key column="p_id"/>
  <index column="indexs" type="integer"/>
  <one-to-many class="Address"/>
</list>
```

用 Map 来配置

```
<map name="addresss" inverse="false" cascade="all">
  <key column="p_id"/>
  <index column="map_key" type="string"/>
  <one-to-many class="Address"/>
</map>
```



t_person @hibernate_db (localhost) - 表

id	t_name	age	birthday
1	tom	18	2013-08-12 15:32:36

t_address @hibernate_db (localhost) - 表

id	descs	codes	p_id	map_key
1	广东广州	500000	1	gd
2	湖南郴州	432000	1	hn

多对多 外键双向关联配置

```
<id name="id"><generator class="native"/></id>
<property name="name" column="t_name" />
<property name="age"/>
<property name="birthday"/>
<!-- 多对多 双向 关联
    多个人对应多个 地址数据双方都使用集合进行存储
    table 表示指定一个关联关系的中间表名称-->
<set name="addresss" cascade="all" inverse="true" table="person_join_address">
    <!-- 指定再中间表中Person的关联关系列名 -->
    <key column="p_id"/>
    <!-- 使用many-to-many 进行两端配置 -->
    <many-to-many class="Address" column="a_id"/>
</set>
</class>

<class name="Address" table="t_address">
    <id name="id"><generator class="native"/></id>
    <property name="descs"/>
    <property name="codes"/>
    <set name="persons" cascade="all" table="person_join_address">
        <key column="a_id"/>
        <many-to-many class="Person" column="p_id"/>
    </set>
</class>
```

保持一致

保持一致

中间表名称要一致
否则会多一个没用的表

1. 多对多要使用中间表，注意对应的那些叉号，**注意 many-to-many**
2. 多对多的时候，所有对另外一个表的查询都是延时查询
3. Table 两个名字可以不一样，最好一样

外键单向关联配置

在主表的配置，外键一定是在从表里，从表对象设置外键映射，才能把主表外键拿到

```
<hibernate-mapping package="org.louis.domain">
  <class name="Member" table="TEST_MEMBER">
    <id name="id" column="ID">
      <generator class="uuid.hex"></generator>
    </id>
    <property name="age" column="AGE"></property>
    <property name="name" column="NAME"></property>
    <!--set元素，就是定义一个集合，它的name属性值是对应的POJO中的相关属性名称-->
    <set name="orders" cascade="all">
      <key column="MEMBER_ID"></key><!--指定“多”的一段的外键，与“一”端得主键相关联-->
      <one-to-many class="Order"/><!--指定了“多”端对应的类-->
    </set>
  </class>
</hibernate-mapping>
```

Order.hbm.xml:

```
<hibernate-mapping package="org.louis.domain">
  <class name="Order" table="TEST_ORDER">
    <id name="id" column="ID">
      <generator class="native"></generator>
    </id>
    <property name="name" column="NAME"></property>
    <property name="num" column="NUM"></property>
    <!--外键-->
    <property name="memberId" column="MEMBER_ID"></property>
  </class>
</hibernate-mapping>
```

继承配置

单表继承

- 1. 通过辨别链区分不同的对象
- 2. 子类与父类配在同一个 XML 文件里
- 3. 注意：辨别列必须紧跟 ID，而且其他子类 subclass 的配置必须在父类 property 的后面

T_PERSON

P_ID INT (NN)

TYPE VARCHAR(255) (NN)

NAME VARCHAR(255)

BIRTHDAY DATETIME

SCORE INT

CLASSS VARCHAR(255)

p_id	type	name	birthday	score	classs
1	org.fkjava.bean.Student 辨别列	admin	2012-12-23 09:47:46	90	J1205
2	org.fkjava.bean.Person	person	2012-12-23 09:50:25	(Null)	(Null)
3	org.fkjava.bean.Student	admin	2012-12-23 09:55:08	(Null)	(Null)

子类的属性

```

<class name="Person" table="t_person" >
  <id name="id" column="p_id">
    <generator class="native"/>
  </id>
  <!-- 单表继承关系，需要添加一个辨别列，对不同实例的数据进行区分，
        辨别列 不需要 我们程序员控制
        type="string" 指定 辨别列 数据类型
        column 辨别列 的 列名
        注意：辨别列 只能在id 属性下面配置
  -->
  <discriminator type="string" column="type"/>
  <property name="name"/>
  <property name="birthday"/>

  <!-- 使用subclass 配置 子类的属性
        name 对应子类类型名
  -->
  <subclass name="Student">
    <!-- 配置子类的属性（配置与普通的属性配置一致） -->
    <property name="score"/>
    <property name="classs"/>
  </subclass>

  <class name="Person" table="t_PERSON" dynamic-update="true" discriminator-value="1">
    <id name="ID">
      <column name="ID" />
      <generator class="native" />
    </id>
    <discriminator type="integer" column="type"/>

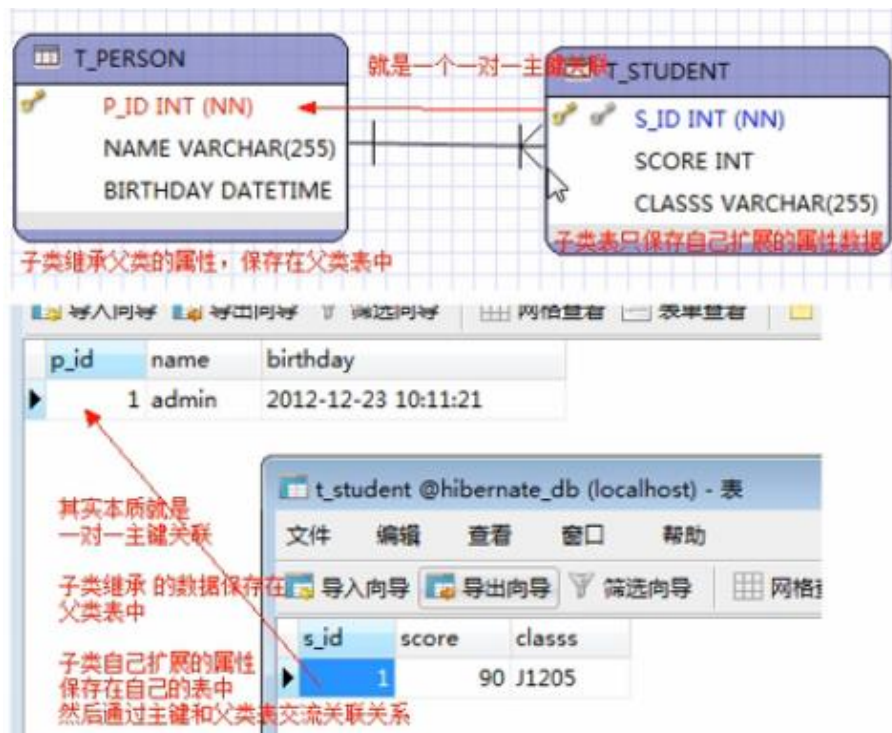
```

通过subclass进行子类的配置
有多个子类，直接按现在的配置复制就可以了

使用 integer 做辨别列，记得加入 discriminator-value

具体表继承

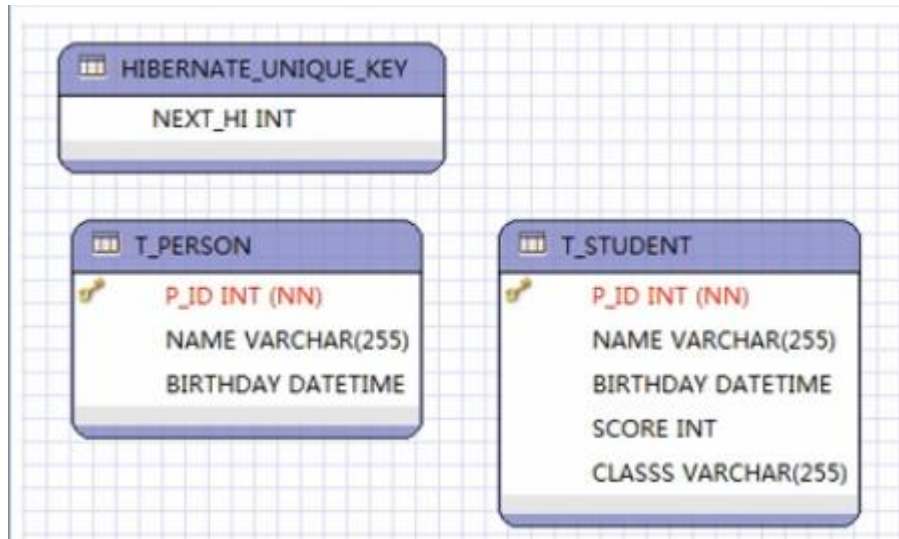
1. 父类的数据存在父类，子类的数据存在子类
2. 一定要有 key



```
<class name="Person" table="t_person" >
  <id name="id" column="p_id">
    <generator class="native"/>
  </id>
  <property name="name"/>
  <property name="birthday"/>
  <!-- 通过joined-subclass 保存关联子类实例 -->
  <joined-subclass name="Student" table="t_student">
    <key column="s_id"/>
    <property name="score"/>
    <property name="classs"/>
  </joined-subclass>
</class>
```

每个具体类一个表

1. 子类和父类的数据都存在子类里，不常用
2. 使用高低值算法
3. 注意：子类不需要配置 key
4. 如果选择插入子类，父类不会有数据



```
<class name="Person" table="t_person" >
  <id name="id" column="p_id">
    <!-- 设置主键生成策略，为高低算法 -->
    <generator class="hilo"/>
  </id>
  <property name="name"/>
  <property name="birthday"/>
  <!-- 通过union-subclass 保存关联子类实例 -->
  <union-subclass name="Student" table="t_student">
    <property name="score"/>
    <property name="classs"/>
  </union-subclass>
</class>
```


HQL

HQL是一种以面向对象的方式来写查询语句结构

HQL中只能有 属性和类

```
select name password from Person where name like %t_%;
```

HQL是通过session.createQuery(hql);返回一个Query对象

```
public void testQuery() {  
    //HQL语句是Hibernate推荐给我们的进行批量数据查询的查询语言  
    //是完全面向对象的方式来进行查询语句的开发  
    //标准的SQL select t_name from t_person  
    //HQL语句 select name from Person  
    String hql = "from Person";  
    //将hql交给session.createQuery(hql)方法, 返回一个Query对象  
    Query query = session.createQuery(hql);  
    //通过query对象的list方法可以获得符合条件的查询结果集  
    List<Person> list = query.list();  
    int count = 0;  
    for(Person p : list){  
        System.out.println(++count+"----"+p);  
    }  
}
```

查全表

list 只支持一级缓存的写入（任何读取都会从数据库读，不去一级缓存读取）

查某个字段

（查询什么字段，就返回什么 list）

```
public void test2() {  
    String hqlString = "select name from Person";  
    Query query = session.createQuery(hqlString);  
    @SuppressWarnings("unchecked")  
    List<String> list = query.list();  
  
    for (String person : list){  
        System.out.println(person);  
    }  
}
```

```

@Test
public void testQuery10() {
    //可以在hql语句上使用构造器的方式来绑定一个查询数据，并将匹配条件的数据返回成对应的对象
    String hql = "from Person";
    Query query = session.createQuery(hql);
    //获得集合中的指定下标位置的一条数据，但是其他满足条件的数据都已经存储在缓存中了
    List<Person> list = query.list();
    Person p = list.get(0);
    System.out.println(p);
    System.out.println("-----");
    Person p2 = (Person) session.get(Person.class, 99);
    System.out.println(p2);
}

```

查多个字段会返回二维对象数组

(查询的是属性名，from 类名， Arrays.toString)

```

List<Object[]> list = query.list();

public void test2() {
    String hqlString = "select name, password from Person";
    Query query = session.createQuery(hqlString);
    @SuppressWarnings("unchecked")
    List list = query.list();

    for (int i = 0; i < list.size(); i++) {
        Object[] objects = (Object[]) list.get(i);
        System.out.println((String) objects[0]);
        System.out.println((int) objects[1]);
    }
}

```

另外一种打印方法，延迟加载，

支持一级缓存的写入和读取：

```

/**
 * iterate 是延迟加载数据
 * n + 1
 * 1 表示先通过一条语句将符合条件的数据的id获取到
 * n 表示 根据id去获取对应的数据，是在真正使用数据的时候才通过语句根据id去获取
 */

```

```

@Test
public void testQuery2() {
    String hql = "from Person";
    Query query = session.createQuery(hql);
    Iterator<Person> it = query.iterate();
    while(it.hasNext()){
        Person p = it.next();
        System.out.println(p);
    }
}

```

使用构造器返回数据

1. new 的对象必须有映射，不能有 null 值

```

@Test
public void testQuery6() {
    //可以在hql语句上使用构造器的方式来绑定一个查询数据，并将匹配条件的数据返回成对应的对象
    String hql = "select new Person(name,password,birthday) from Person";
    Query query = session.createQuery(hql);

    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}

```

使用 List 返回数据

(注意 new List 和 List<List>)

```

@Test
public void testQuery7() {
    //可以在hql语句上使用List的方式来绑定一个查询数据，并将匹配条件的数据返回成对应的List集合
    String hql = "select new List(name,password,birthday) from Person";
    Query query = session.createQuery(hql);

    List<List> list = query.list();
    for(List l : list){

        for(int i=0;i<l.size();i++){
            System.out.println(l.get(i));
        }
        System.out.println("-----");
    }
}

```

唯一的结果

如果返回的不是 **unique** 结果，会报错

```
@Test
public void testQuery11() {
    String hql = "from Person where id = 99";
    Query query = session.createQuery(hql);
    //如果明确指定获取的数据是唯一的一条记录，我们可以使用uniqueResult()来获取，
    //如果不是唯一的数据将报异常
    Person p = (Person) query.uniqueResult();
    System.out.println(p);
}
```

分页查询

第一个数是开始（从下一个开始），第二个数是查几条

```
@Test
public void testQuery12() {
    String hql = "from Person";
    Query query = session.createQuery(hql);
    //分页操作
    //分页的起始位置(会从指定位置的下一个位置开始)
    query.setFirstResult(0);
    //每次获取数据的条数
    query.setMaxResults(10);
    List<Person> list = query.list();
    for(Person p : list) {
        System.out.println(p);
    }
}
```

单参数绑定

```
@Test
public void testQuery14(){
    //参数绑定
    String hql = "from Person where id = ?";
    Query query = session.createQuery(hql);
    //使用? 占位符进行参数绑定
    //JDBC的? 占位符进行参数绑定的时候是从 1 开始
    //HQL的? 占位符进行参数绑定的时候是从 0 开始
    //setXxx();参数1: ?占位符位置,参数2: 实际参数
    query.setInteger(0, 99);
    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}
```


多参数占位符绑定的三种方法

```
public void testQuery15(){
    //参数绑定
    String hql = "from Person where id = ? and name = ?";
    Query query = session.createQuery(hql);
    query.setInteger(0, 99);
    query.setString(1, "admin-98");
    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}

@Test
public void testQuery16(){
    //参数绑定
    String hql = "from Person where name = ?8 and id = ?1 ";
    Query query = session.createQuery(hql);
    //参数的位置使用?后面的数字进行确定,但是字符串类型的数字值
    query.setInteger("1", 99);
    query.setString("2", "admin-98");
    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}

@Test
public void testQuery17(){
    //命名参数绑定
    String hql = "from Person where name = :name and id = :id ";
    Query query = session.createQuery(hql);

    query.setInteger("id", 99);
    query.setString("name", "admin-98");
    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}
```

第一种: 顺序占位符

第二种: 数字占位符, 但是 set 的时候需要用数字字符串 (Hibernate 4 扩展)

第三种: 参数命名绑定, 一定要加冒号

批量数据绑定

可以用 Object 数组也可以用 List<Integer>

```
@Test
public void testQuery18() {
    //String hql = "from Person where id in(3,6,9)";
    String hql = "from Person where id in(:ids)";
    Query query = session.createQuery(hql);
    //批量参数绑定可以使用数组
    //query.setParameterList("ids", new Object[]{3,6,9});
    //批量参数绑定可以使用集合
    List<Integer> ids = new ArrayList<>();
    ids.add(3);
    ids.add(6);
    ids.add(9);
    query.setParameterList("ids", ids);

    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}
```

模糊查询

```
@Test
public void testQuery19() {
    //String hql = "from Person where name like '%-99%'";
    String hql = "from Person where name like :name";
    Query query = session.createQuery(hql);
    //模糊查询，在进行参数绑定设置的时候不需要加单引号
    query.setString("name", "%-99%");
    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}
```

联合查询

```
@Test
public void test4() {
    String hqlString = "SELECT p, a FROM Person p INNER JOIN p.addressesSet a WHERE p.ID = a.person";
    Query query = session.createQuery(hqlString);

    List<Object[]> objects = query.list();

    for (Object[] objects2 : objects) {
        System.out.println(Arrays.toString(objects2));
    }
}
```

1. 可以使用内外左右连接
2. 记得一定要配置路径! p.addressesSet

HQL 更新

```
public void testUpdate() {
    String hql = "update Person set password=1";
    tx = session.beginTransaction();
    //批量更新数据
    int count = session.createQuery(hql).executeUpdate();
    tx.commit();
    String msg = count>0?"成功["+count+"]":"失败";
    System.out.println("更新--"+msg);
}
```

1. 应该配置 Person 而不是表名
2. 涉及数据库的更改应该用 transaction
3. 删除外键不能使用 HQL
4. Flush 并不影响缓存, 不能用来同步; 但是可以使用 clear 重新加载缓存

5. HQL 不影响缓存, 不能同步更新到缓存→
不一致

持久化配置 HQL Query

```
/**
 * 在hibernate开发中推荐大家将HQL和标准的SQL查询语句尽量配置的
 * 持久化映射文件中 (*.hbm.xml)
 * <query name="myQuery">
 *     from Person
 * </query>
 * 然后在代码中使用session.getNamedQuery("myQuery");获取对应的Query对象
 */
@Test
public void testQuery1(){
    Query query = session.getNamedQuery("myQuery");
    List<Person> list = query.list();
    for(Person p :list ){
        System.out.println(p);
    }
}
```

```
<query name="myQuery2">
    from Person
</query>
```

```
<query name="myQuery3">
    <!-- 如果遇到大于或者小于可以使用如下方式进行转义
        &lt; = <
        &gt; = >
    -->
    from Person where id &lt; 5
</query>
```

```
<query name="myQuery4">
    <!-- 如果遇到一些特殊符号可以使用如下方式进行转义 -->
    <![CDATA[from Person where id < :id]]>
</query>
```

记得关联 query.xml

QBC

Qbc 没有 iterator，只有 list

QBC

Hibernate提供了直观的Criteria查询API，Query By Criteria简称QBC。

条件查询

通过API的方式绑定查询条件和排序。

好处是：

我们可以在不会SQL的情况下进行Hibernate的 查询

基础用法

```
@Test
public void testQuery() {
    //通过session获得createCriteria获得Criteria对象
    //给定的参数是需要查询的持久化类的class
    Criteria query = session.createCriteria(Person.class);
    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}
```


进阶用法，包括加限制，加顺序

```
@Test
public void testQuery2() {
    Criteria query = session.createCriteria(Person.class);
    //query.add(Restrictions.le("id", 5));
    //query.add(Restrictions.eq("id", 5));
    //query.add(Restrictions.like("name", "%-8%"));
    //query.add(Restrictions.between("password", 125, 128));
    //query.add(Restrictions.in("id", new Object[]{3,6,9}));
    //指定结果集排序
    //query.addOrder(Order.asc("id"));
    query.addOrder(Order.desc("id"));

    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}

/**
 * 通过Property类的静态方法进行查询条件的绑定
 * 其实底层就是对Restrictions的封装
 */
@Test
public void testQuery3() {
    Criteria query = session.createCriteria(Person.class);
    //query.add(Property.forName("id").le(5));
    query.add(Property.forName("password").between(125, 128));

    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}
```

样例查询

```
/**
 * 样例查询
 * 根据给定的对象的设置了有值的属性来进行查询
 * 会将有值的属性获取拼装成sql语句的查询条件进行查询
 * 注意:如果一个对象的属性是基本类型,那么因为有默认值,所以也会获取拼装
 */
@Test
public void testQuery() {
    Person p = new Person();
    p.setName("admin-8");
    p.setPassword(131);
    Criteria query = session.createCriteria(Person.class);
    query.add(Example.create(p));
    List<Person> list = query.list();
    for (Person pp : list) {
        System.out.println(pp);
    }
}
```

离线查询

```
/**
 * 离线查询
 * 可以在绑定查询数据和条件之前不需要session (最晚打开session)
 * 当真正使用的时候才传递一个当前的session过来就可以了
 *
 * session是使用的一些技巧:
 * 最晚打开
 * 最早关闭
 * 不要长时间打开
 */
@Test
public void testQuery() {
    //通过DetachedCriteria绑定需要查询的持久化类
    DetachedCriteria dc = DetachedCriteria.forClass(Person.class);
    //绑定查询条件
    dc.add(Restrictions.between("id", 6, 9));
    //在使用的时候绑定一个session就可以了
    Criteria query = dc.getExecutableCriteria(session);
    List<Person> list = query.list();
    for (Person pp : list) {
        System.out.println(pp);
    }
}
```

SQL 查询

SQL

就是以前学习的JDBC的标准SQL语句
开发非常方便的使用数据库的特性

SQL 语句方式适合在
一个已经存在的项目的基础上进行优化和扩展

不适合新程序

基础应用

```
@Test
public void testQuery() {
    //基于标准的sql语句进行查询
    String sql = "select * from t_person";
    //通过session的createSQLQuery获得一个SQLQuery
    //通过源代码我们发现SQLQuery就是Query的子类
    SQLQuery query = session.createSQLQuery(sql);
    //因为是基于表的操作，所以返回的数据是标准值的一个Object数组
    List<Object[]> list = query.list();
    for(Object[] objs : list){
        System.out.println(Arrays.toString(objs));
    }
}
```

绑定持久化类，直接返回对象

1. 使用 addEntity()

```

@Test
public void testQuery2() {

    String sql = "select * from t_person";
    SQLQuery query = session.createSQLQuery(sql);
    //指定查询结果与某个持久化类进行绑定，返回的结果就是对象了
    query.addEntity(Person.class);
    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}

```

2. 绑定参数

```

@Test
public void testQuery3() {

    String sql = "select * from t_person where id in(:ids)";
    SQLQuery query = session.createSQLQuery(sql);
    //指定查询结果与某个持久化类进行绑定，返回的结果就是对象了
    query.addEntity(Person.class);
    //绑定参数
    query.setParameterList("ids", new Object[]{3,6,9});
    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}

```

3. 持久化配置，记得强行转化成 SQLQuery

```

@Test
public void testQuery4() {
    //根据sql语句在配置文件的名字获得语句
    SQLQuery query = (SQLQuery) session.getNamedQuery("mysqlquery");
    query.addEntity(Person.class);
    query.setInteger("id", 8);
    Person p = (Person) query.uniqueResult();
    System.out.println(p);
}

```


数据抓取

数据抓取策略

当程序在实体对象图的关联关系间进行导航的时候，怎样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）中声明。

三种抓取策略：

连接抓取(Join fetching) Hibernate通过 在SELECT语句使用外连接(outer join)来获得的关联实例或者关联集合。

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。除非显式的指定lazy="false"禁止 延迟抓取，否则只有当真正访问关联关系的时候，才会执行第二条select语句。 **默认配置**

子查询抓取(Subselect fetching) 另外发送一条SELECT 语句抓取在前面查询到的所有实体对象的关联集合。除非显式指定lazy="false"，否则只有当真正访问关联关系的时候，才会执行第二条select语句。 **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。

<!-- fetch 集合数据的抓取策略

select 默认配置，会在实际使用关联关系的数据的时候，通过一条新的语句获取关联关系所对应的实例，除非lazy=false

join 和使用left outer join 连接将关联关系在一条语句中查询出来

subselect 会在实际使用关联关系的数据的时候，通过一条新的语句获取关联关系所对应的实例，除非lazy=false

batch-size 配置大批量数据 的 分批 获取 策略

例如 有10条记录，我们的batch-size设置为 3

那么hibernate将 通过 4条语句分批获取这10条记录

分批策略是 3 3 3 1]

-->

```
<set name="addressSet" cascade="all" inverse="true" fetch="join">
```

```
<set name="addressSet" cascade="all" inverse="true" batch-size="3">
```

- Join fetching: Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.
- Select fetching: a second SELECT is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.
- Subselect fetching: a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.

- Batch fetching: an optimization strategy for select fetching. Hibernate retrieves a batch of entity instances or collections in a single SELECT by specifying a list of primary or foreign keys.

锁机制

锁机制

乐观锁:

机制采取了更加宽松的加锁机制。乐观锁机制多数基于程序中的数据存储逻辑，也有一定的局限性，例如由于乐观锁机制是在A系统中实现，那么来自B系统的更新操作将不受A系统的控制，因此可能会造成脏数据被更新到数据库中。

悲观锁:

多数依靠底层数据库的锁机制实现，以保证操作最大程度的独占性。但数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。

乐观锁

为数据增加版本标识，在基于表的版本解决方案中，一般通过为表增加一个“version”字段来实现。读取数据时，将此version一起获取，之后更新时，对version+1。将提交数据的version与表中对应记录的当前version进行比较，如果提交的数据version大于表当前version，则予以更新，否则认为是过期数据，更新失败。

<!-- 乐观锁配置

optimistic-lock="version" 配置使用版本号的，方式进行乐观锁控制
并且需要在持久化类上添加一个字段private int version

-->

```
<class name="Person" table="t_person" optimistic-lock="version">
  <id name="id">
    <generator class="native"/>
  </id>

  <version name="version" type="integer"/>

  <property name="name" column="t_name"/>
  <property name="password"/>
  <property name="birthday"/>
```

I

Hibernate 会自动更新数据，更新失败指的是抛出异常

```
/**
 * hibernate在获取数据的时候返回一个锁状态
 * 在提交数据的时候会动的将锁状态进行改变
 * 如果提交的数据锁状态小于表字段的的状态，将不允许提交更新数据，可以有效的防止脏数据
 */
```

过滤机制

1. 定义过滤器

```
<!-- 定义过滤器，并指定参数 -->
<filter-def name="password_filter">
    <filter-param name="ps" type="integer"/>
    <filter-param name="ps2" type="integer"/>
</filter-def>
```

Condition 对应列名，不是属性名

Type 一定得 **小写**

2. 开启过滤器

```
<!-- 引入使用指定的过滤器 -->
<filter name="password_filter" condition="password between :ps and :ps2"/>

@Test
public void testQuery() {
    //启用指定名称对应的过滤器
    Filter passwordFilter = session.enableFilter("password_filter");
    //过滤器参数绑定（过滤条件绑定）
    passwordFilter.setParameter("ps", 125);
    passwordFilter.setParameter("ps2", 129);

    Query query = session.createQuery("from Person");
    List<Person> list = query.list();
    for(Person p : list){
        System.out.println(p);
    }
}
```

3. 关闭过滤器

```
//停用指定的过滤器
//如果在一个有开启过滤器的session中进行对应的持久化类的获取操作都会自动加上过滤器的过滤条件
session.disableFilter("password_filter");

session.disableFilter("password_filter");
```

拦截器

1. 可以直接 override EmptyInterceptor 基类，也可以继承 Interceptor 接口

```
/**
 * 自定义拦截器，实现在指定方法执行之前插入一个自己的业务逻辑代码
 *
 * 如果实现的是Interceptor 接口，那么需要重写全部的抽象方法
 * 所以推荐大家使用继承EmptyInterceptor 父类，然后想重写什么方法就重写什么方法，因为他已经帮助我们做了空实现
 *
 * 注意一定要调用super.onXXX
 *
 * 可以通过参考Interceptor 的API或者源代码了解其他的操作正确的拦截器实现方法是什么。
 */
system.out.println("state:"+Arrays.toString(state));
System.out.println("propertyNames:"+Arrays.toString(propertyNames));
System.out.println("types:"+Arrays.toString(types));

if(entity.getClass() == Person.class){
    for(int i =0;i<propertyNames.length;i++){
        if(propertyNames[i].equals("birthday")){
            if(state[i] == null){
                Person p = (Person)entity;
                p.setBirthday(new Date());
            }
        }
    }
}

return super.onSave(entity, id, state, propertyNames, types);
```

2. 配置并开启拦截器

```
public void setUp() throws Exception {
    System.out.println("-----测试前初始化-----");
    Configuration cfg = new Configuration().configure();
    //全局拦截器配置
    //cfg.setInterceptor(new MyInterceptor());

    ServiceRegistry sr = new ServiceRegistryBuilder().applySettings(
        cfg.getProperties()).buildServiceRegistry();
    factory = cfg.buildSessionFactory(sr);
    //局部拦截器 配置
    session
    =factory.withOptions().interceptor(new MyInterceptor()).openSession();
}
```

缓存

一级缓存

1-一级缓存

一级缓存就是 session 级别的缓存

session会将与自己关联的实体进行缓存，缓存在自己持有的一级缓存中
session操作的 增删改查

一级缓存和session生命周期 一致

session关闭了，一级缓存就没有了

一级缓存中的数据不能在多个session中共享， 就可以理解为 局部缓存

-----必须掌握-----

get load iterate

支持一级缓存 读写 操作

list

只支持一级缓存 写 操作

get list 是 立即 查询

load iterate 是 延迟 查询

一级缓存管理

evict(Object obj) 将指定的持久化对象从一级缓存中清除，释放对象所占用的内存资源，指定对象从持久化状态变为脱管状态，从而成为游离对象。

clear() 将一级缓存中的所有持久化对象清除，释放其占用的内存资源。

contains(Object obj) 判断指定的对象是否存在于一级缓存中。

flush() 刷新一级缓存区的内容,使之与数据库数据保持同步。

```
//session = factory.openSession();
```

```
//在多线程 环境下进行Hibernate的开发，建议大家使用getCurrentSession()获得session比较安全
```

```
//使用session需要注意的问题：
```

```
//最晚打开（要用的时候才开启）
```

```
//尽早关闭（用完立刻关闭）
```

```
//不要操作太长时间或者跨用户操作
```

```
//如果使用getCurrentSession()获得session需要
```

```
//hibernate.cfg.xml文件中进行配置<property name="hibernate.current_session_context_class">thread</property>
```

```
//注意使用getCurrentSession获得的session 不能 手动关闭
```

```
//还需要开启事务的支持（查询也需要）
```

```
session = factory.getCurrentSession();
```



```
<!-- 开启线程安全的session -->
<property name="hibernate.current_session_context_class">thread</property>
```

线程安全的 session 一定要开事务

openSession 适用于能快速完成的请求

currentSession 适用于需要几个连续的步骤才能完成的。

```
@Test
public void testGet(){
    //如果获得的是一个线程安全的session注意在进行查询的时候也要事务的支持
    tx = session.beginTransaction();
    Person p = (Person)session.get(Person.class,14);
    System.out.println(p);
    tx.commit();
}
```

As explained in this forum [post](#), 1 and 2 are related. If you set `hibernate.current_session_context_class` to `thread` and then implement something like a servlet filter that opens the session - then you can access that session anywhere else by using the `SessionFactory.getCurrentSession()`.

`SessionFactory.openSession()` always opens a new session that you have to close once you are done with the

operations. `SessionFactory.getCurrentSession()` returns a session bound to a context - you don't need to close this.

If you are using Spring or EJBs to manage transactions you can configure them to open / close sessions along with the transactions.

You should never use "one session per web app" - session is not a thread safe object - cannot be shared by multiple threads. You should always use "one session per request" or "one session per transaction"

二级缓存，以 EHcache 为例

二级缓存 = sessionFactory 相关 生命周期与sessionFactory一致

load get iteratr 支持 二级缓存 读写 操作

list 支持二级缓存的 写 操作

缓存 对象实例， 不缓存属性

二级缓存 需要配置第三方插件

1. Ehcache.xml

```
<ehcache>
  <!-- 指定当缓存需要写出数据的时候数据保存磁盘位置 -->
  <diskStore path="c:/cache_temp"/>
  <!--
    maxElementsInMemory - 允许在二级缓存中的持久化对象数据量
    eternal - 缓存中的对象是否允许销毁 false 表示可以销毁
    timeToLiveSeconds - 缓存中对象的激活时间
    timeToIdleSeconds - 当持久化对象激活时间到期以后，还可以存活多长时间（钝化时间）
    overflowToDisk - 是否允许缓存数据序列化到磁盘
  -->
  <defaultCache
    maxElementsInMemory="5"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"/>
</ehcache>
```

2. Hibernate.cfg.xml

```
<!-- 开启二级缓存（默认是开启状态） -->
<property name="hibernate.cache.use_second_level_cache">true</property>
<!-- 配置二级缓存的实现类（第三方插件包） -->
<property name="hibernate.cache.region.factory_class">org.hibernate.cache.internal.EhCacheRegionFactory</property>
<!-- 配置二级缓存的持久化类 -->
<property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
```

3. 开启持久化类的二级缓存

```
<!-- 开启持久化类的二级缓存配置
配置方式有2种：
1：可以在持久化配置文件中配置<class ...><cache usage="read-write"/>...</class>
2：可以在hibernate.cfg.xml文件中进行统一配置（推荐）
-->
<class-cache usage="read-write" class="org.fkjava.pojo.Person"/>
```

4. 使用二级缓存

```

/**
 * 在有配置二级缓存的查询中步骤如下：
 * 先搜索 一级缓存--> 二级缓存--->SQL查询（数据库查询）--->将数据写入缓存中
 *
 * 二级缓存是可以在多个session中共享
 *
 */
@Test
public void testQuery() {
    session = factory.openSession();

    Person p = (Person) session.get(Person.class, 1);
    System.out.println(p);
    session.close();
    System.out.println("-----");
    session = factory.openSession();
    Person p2 = (Person) session.get(Person.class, 1);
    System.out.println(p2);

    session.close();
}

```

二级缓存的管理

```

public void testQuery() throws InterruptedException{
    String hql = "from Person";
    session = factory.openSession();
    //获得二级缓存操作 对象 句柄
    Cache cache = factory.getCache();
    Query query = session.createQuery(hql);
    Iterator<Person> it = query.iterate();
    while(it.hasNext()){
        System.out.println(it.next());
    }
    System.out.println("-----");
    boolean flag = cache.containsEntity(Person.class, 9);
    System.out.println("指定的持久化对象是否在二级缓存中: "+flag);
    //将一个持久化类踢出二级缓存
    //cache.evictEntity(Person.class, 9);
    cache.evictEntityRegion(Person.class);
    System.out.println("是否存在:"+cache.containsEntity(Person.class, 9));
}

```

二级缓存的适用范围

二级缓存管理

不适合加载到二级缓存中的情况:

1. 经常被修改的数据
2. 绝对不允许出现并发访问的数据
3. 与其他应用共享的数据

合适加载到二级缓存中的情况:

1. 数据更新频率低
2. 允许偶尔出现并发问题的非重要数据
3. 不会被并发访问的数据
4. 常量数据
5. 不会被第三方修改的数据

查询缓存，基于二级缓存

查询缓存 不好

查询缓存 需要依赖 二级缓存

对于经常使用的查询语句，如果启用了查询缓存，当第一次执行查询语句时，Hibernate会把查询结果存放在二级缓存中。以后再次执行该查询语句时，只需从缓存中获得查询结果，从而提高查询性能。

使用查询缓存需要注意：

只能对于 不变化的语句有效或者不变的数据库表数据

select p from Person;

list 支持查询缓存

iterate 不支持

但是我们在开发中很少使用不变化的语句，所以查询缓存用处不大。

1. Hibernate.cfg.xml

```
<!-- 开启查询缓存（默认是关闭状态）
      查询缓存是基于二级缓存的
-->
<property name="hibernate.cache.use_query_cache">true</property>
```

2. 在程序中

```
//告诉hibernate先到查询缓存中搜索有没有相同的查询语句查询过，有就将之前查询的数据直接返回
query.setCacheable(true);
```

Hibernate 最佳实践

Hibernate最佳实践

1、使用Configuration装载映射文件时，不要使用绝对路径装载。最好的方式是通过`getResourceAsStream()`装载映射文件，这样Hibernate会从classpath中寻找已配置的映射文件。

2、SessionFactory的创建非常消耗资源，整个应用一般只要一个SessionFactory就够了，只有多个数据库的时候才会使用多个SessionFactory。

3、在整个应用中，Session和事务应该能够统一管理。（Spring为Hibernate提供了非常好的支持）

4、将所有的集合属性配置设置为懒加载（`lazy=true`）

5、在定义关联关系时，集合首选Set，如果集合中的实体存在重复，则选择List（在定义配置文件时，可以将List定义为idbag），数组的性能最差。

6、在一对多的双向关联中，一般将集合的inverse属性设置为true，让集合的对方维护关联关系。

例如Person-Address，由Address来维护Person和Address的关联关系。

让拥有外键的一端来进行关联关系的维护比较好

7、在执行更新的时候尽量配置 `dynamic-update="true"`，表示没有修改的属性不参与更新操作

8、HQL子句本身大小写无关，但是其中出现的类名和属性名必须注意大小写区分。

9、如果可以，使用乐观锁代替悲观锁

10、如果要很好的掌握Hibernate，熟练掌握关系数据库理论和SQL是前提条件。