

什么是 Hibernate ?

- 一个框架
- 一个 Java 领域的持久化框架
- 一个 ORM 框架

对象的持久化

- 狭义的理解，“持久化”仅仅指把对象永久保存到数据库中
- 广义的理解，“持久化”包括和数据库相关的各种操作：
 - 保存：把对象永久保存到数据库中。
 - 更新：更新数据库中对象(记录)的状态。
 - 删除：从数据库中删除一个对象。
 - 查询：根据特定的查询条件，把符合查询条件的一个或多个对象从数据库加载到内存中。
 - **加载**：根据特定的**OID**，把一个对象从数据库加载到内存中。



为了在系统中能够找到所需对象，需要为每一个对象分配一个唯一的标识号。在关系数据库中称之为主键，而在对象术语中，则叫做对象标识(Object identifier-OID).

- 当我们工作在一个面向对象的系统中时，存在一个对象模型和关系数据库不匹配的问题。RDBMSs 用表格的形式存储数据，然而像 Java 或者 C# 这样的面向对象的语言它表示一个对象关联图。

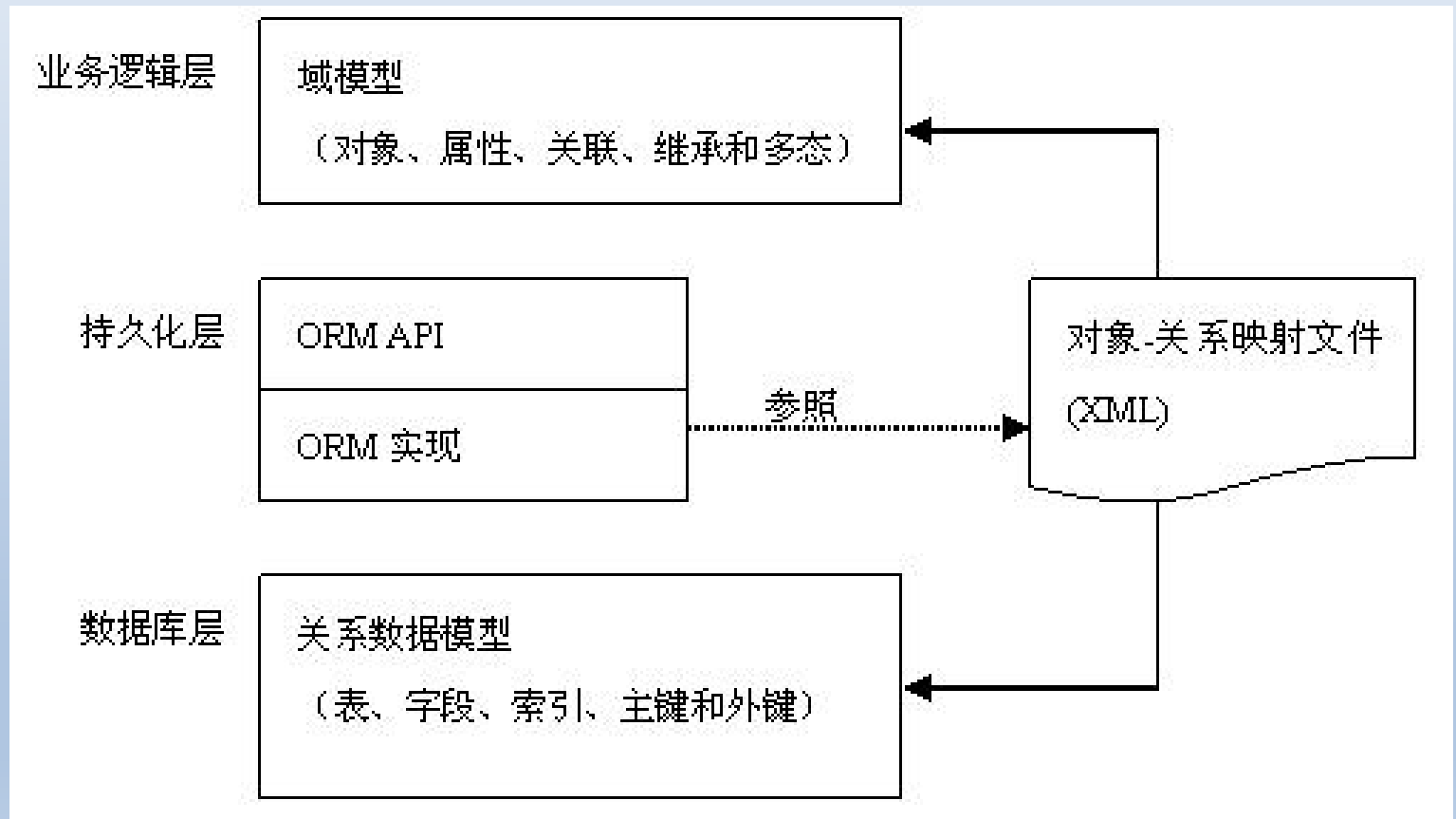
ORM

- ORM(Object/Relation Mapping): 对象/关系映射
 - ORM 主要解决对象-关系的映射

面向对象概念	面向关系概念
类	表
对象	表的行(记录)
属性	表的列(字段)

- ORM的思想：将关系数据库中表中的记录映射成为对象，以对象的形式展现，程序员可以把对数据库的操作转化为对对象的操作。
- ORM 采用元数据来描述对象-关系映射细节，元数据通常采用 XML 格式，并且存放在专门的对象-关系映射文件中。

ORM





Hibernate 优势

Hibernate 使用 XML 文件来处理映射 Java 类别到数据库表格中，并且不用编写任何代码。
为在数据库中直接储存和检索 Java 对象提供简单的 APIs。
如果在数据库中或任何其它表格中出现变化，那么仅需要改变 XML 文件属性。
抽象不熟悉的 SQL 类型，并为我们提供工作中所熟悉的 Java 对象。
Hibernate 不需要应用程序服务器来操作。
操控你数据库中对象复杂的关联。
最小化与访问数据库的智能提取策略。
提供简单的数据询问。

流行的ORM框架

- **Hibernate:**
 - 非常优秀、成熟的 ORM 框架。
 - 完成对象的持久化操作
 - Hibernate 允许开发者采用面向对象的方式来操作关系数据库。
 - 消除那些针对特定数据库厂商的 SQL 代码
- myBatis :
 - 相比 Hibernate 灵活高，运行速度快
 - 开发速度慢，不支持纯粹的面向对象操作，需熟悉sql语句，并且熟练使用sql语句优化功能
- TopLink
- OJB

Hibernate 与 Jdbc 代码对比

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

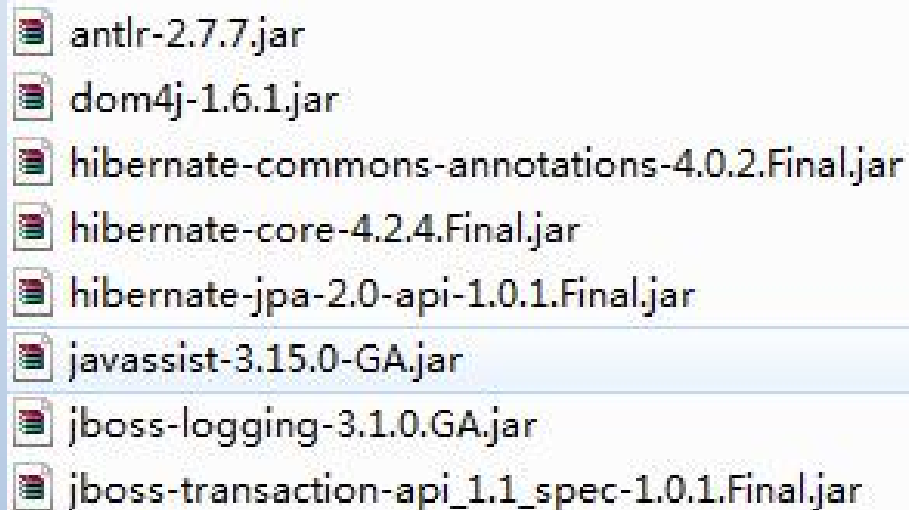
```
public void save(Connection conn, Message m) {  
    PreparedStatement ps = null;  
    String sql = "insert into message values (?, ?)";  
  
    try {  
        ps = conn.prepareStatement(sql);  
        ps.setString(1, m.getTitle());  
        ps.setString(2, m.getContent());  
        ps.execute();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

Hibernate 实现

JDBC 实现

准备 Hibernate 环境

- 导入 Hibernate 必须的 jar 包:



A screenshot of a file explorer window displaying a list of JAR files required for Hibernate. The files are listed in a single column, each preceded by a small icon representing a JAR file. The list includes:

- antlr-2.7.7.jar
- dom4j-1.6.1.jar
- hibernate-commons-annotations-4.0.2.Final.jar
- hibernate-core-4.2.4.Final.jar
- hibernate-jpa-2.0-api-1.0.1.Final.jar
- javassist-3.15.0-GA.jar
- jboss-logging-3.1.0.GA.jar
- jboss-transaction-api_1.1_spec-1.0.1.Final.jar

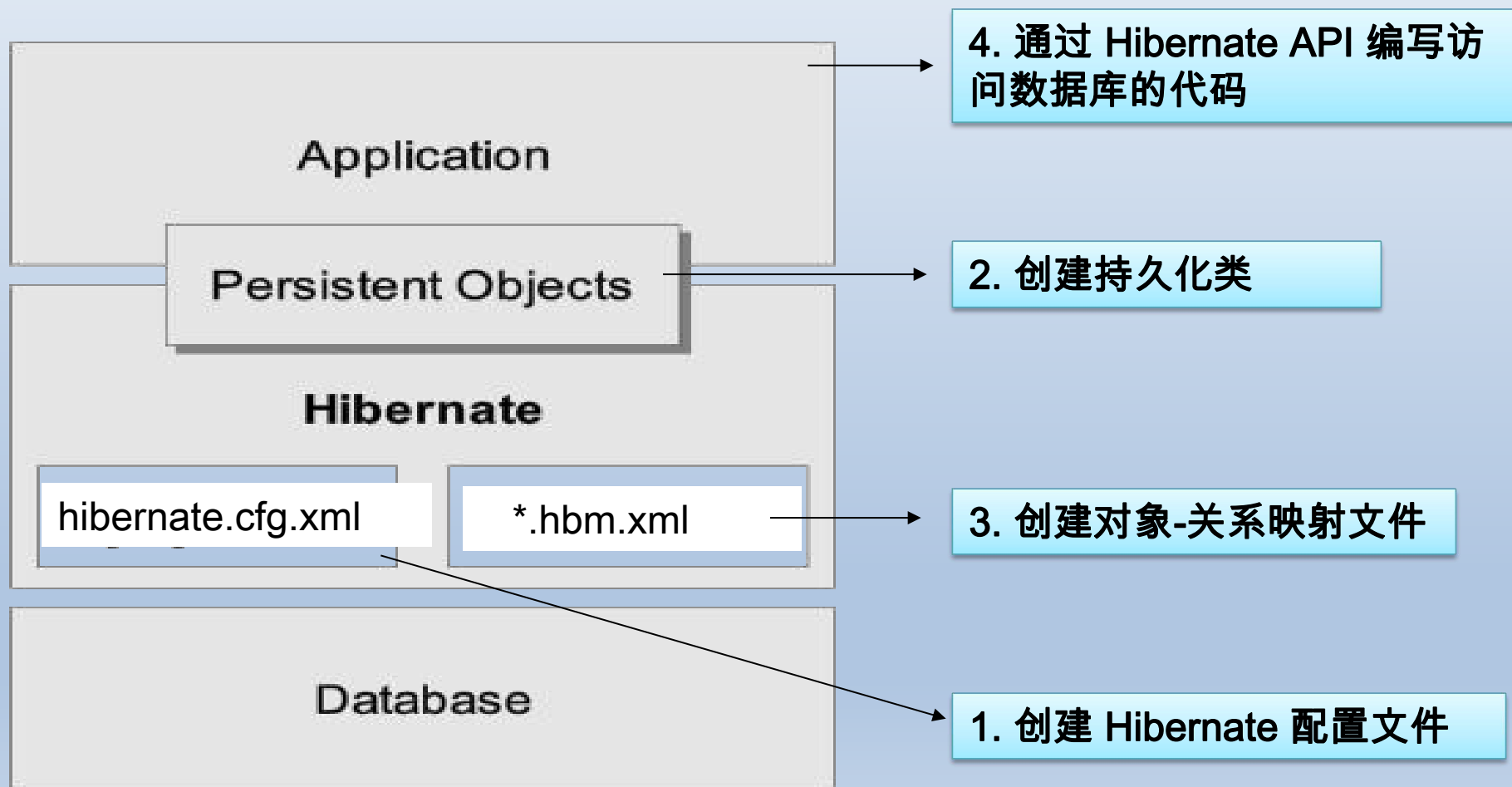
- 加入数据库驱动的 jar 包 :



A screenshot of a file explorer window displaying a single JAR file required for the database driver. The file is listed in a single column, preceded by a small icon representing a JAR file. The file is:

- mysql-connector-java-5.1.7-bin.jar

Hibernate开发步骤



1. 创建持久化 Java 类

- **提供一个无参的构造器**:使Hibernate可以使用 `Constructor.newInstance()` 来实例化持久化类
- **提供一个标识属性(identifier property)**: 通常映射为数据库表的主键字段. **如果没有该属性, 一些功能将不起作用**, 如:
`Session.saveOrUpdate()`
- **为类的持久化类字段声明访问方法(get/set)**: Hibernate对 JavaBeans 风格的属性实行持久化。
- **使用非 final 类**: 在运行时生成代理是 Hibernate 的一个重要的功能. 如果持久化类没有实现任何接口, Hibernate 使用 CGLIB 生成代理. 如果使用的是 final 类, 则无法生成 CGLIB 代理.
- **重写 equals 和 hashCode 方法**: 如果需要把持久化类的实例放到 Set 中(当需要进行关联映射时), 则应该重写这两个方法

1. 创建持久化 Java 类

- Hibernate 不要求持久化类继承任何父类或实现接口，这可以保证代码不被污染。这就是Hibernate被称为低侵入式设计的原因

```
public class News {  
    private Integer id; //field  
    private String title;  
    private String author;  
    private String desc;  
    private String content;  
    private Blob picture;  
    private Date date;  
  
    ....  
}
```

2. 创建对象-关系映射文件

- Hibernate 采用 XML 格式的文件来指定对象和关系数据之间的映射. 在运行时 Hibernate 将根据这个映射文件来生成各种 SQL 语句
- 映射文件的扩展名为 .hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.atguigu.hibernate.helloworld">
<class name="News" table="NEWS" dynamic-insert="true">
```

指定类和表的映射

```
    <id name="id" type="java.lang.Integer">
        <column name="ID" />
        <generator class="native" />
    </id>
    <property name="title" not-null="true" unique="true" length="50"
        type="java.lang.String" column="TITLE" >
```

映射类的属性和表的字段

```
    </property>
    <property name="author" type="java.lang.String">
        <column name="AUTHOR" />
    </property>
    <property name="date" type="date">
        <column name="DATE" />
    </property>
    <property name="desc"
```

指定持久化类的OID
和表的主键的映射

```
        formula="(SELECT concat(title, ',', author) FROM NEWS n WHERE n.id = id)"></property>
```

```
    <property name="content">
```

```
        <column name="CONTENT" sql-type="text"></column>
```

```
    </property>
```

```
    <property name="picture" column="PICTURE" type="blob"></property>
```

```
</class>
```

指定对象标识符生成器, 负责
为 OID 生成唯一标识符

3. 创建 Hibernate 配置文件

- Hibernate 从其配置文件中读取和数据库连接的有关信息，这个文件应该位于应用的 classpath 下。

```
<property name="connection.username">root</property>
<property name="connection.password">1230</property>
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql:///hibernate5</property>
```

指定连接数据库的基本属性信息

指定数据库所使用的 SQL 方言

```
<property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>
```

指定程序运行时是否在控制台输出 SQL 语句

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

指定是否对输出 SQL 语句进行格式化

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

指定程序运行时是否在数据库自动生成数据表

```
<property name="hbm2ddl.auto">update</property>
```

指定程序需要关联的映射文件

```
<mapping resource="com/atguigu/hibernate/helloworld/News.hbm.xml"/>
```

4. 通过 Hibernate API 编写访问数据库的代码

- 测试代码

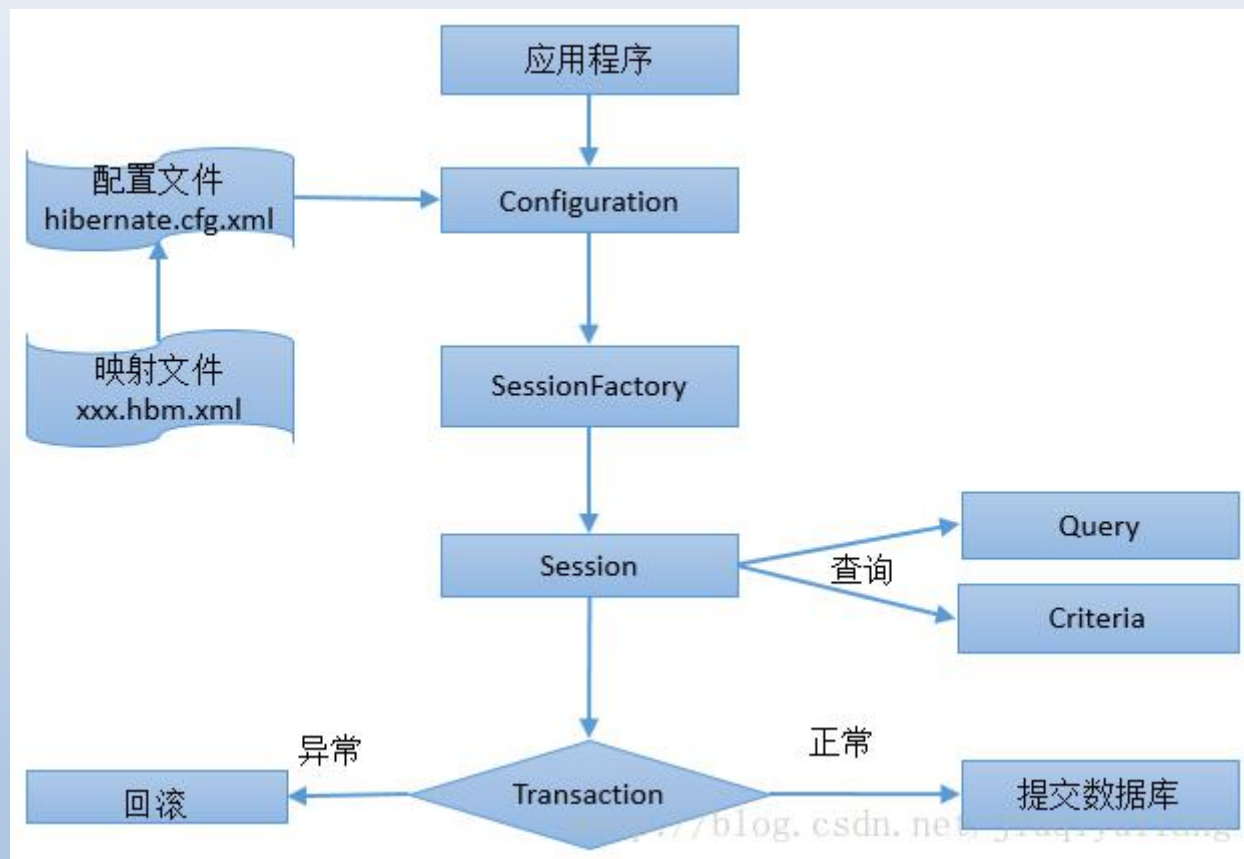
```
SessionFactory sessionFactory = null;  
Configuration configuration = new Configuration().configure();  
sessionFactory = configuration.buildSessionFactory();  
ServiceRegistry serviceRegistry =  
    new ServiceRegistryBuilder().applySettings(configuration.getProperties())  
                                .buildServiceRegistry();  
sessionFactory = configuration.buildSessionFactory(serviceRegistry);  
Session session = sessionFactory.openSession();  
Transaction transaction = session.beginTransaction();  
News news = new News("abc", "gdf", new Date(new java.util.Date().getTime()));  
session.save(news);  
transaction.commit();  
session.close();  
sessionFactory.close();
```

- 下边是控制台输出的 SQL

```
insert into CUSTOMERS (CUSTOMER_NAME, CUSTOMER_ADDRESS, CUSTOMER_PHONE, CUSTOMER_EMAIL, CUSTOMER_PASSWORD)
```

```
十月 30, 2024 3:30:58 下午 org.hibernate.tool.hbm2ddl.SchemaUpdate e  
ERROR: HHH000388: Unsuccessful: alter table hobby_tab add  
十月 30, 2024 3:30:58 下午 org.hibernate.tool.hbm2ddl.SchemaUpdate e  
ERROR: Table 'test.hobby_tab' doesn't exist  
十月 30, 2024 3:30:58 下午 org.hibernate.tool.hbm2ddl.SchemaUpdate e  
INFO: HHH000232: Schema update complete  
Hibernate:  
insert  
into  
    NEWS  
    (TITLE, AUTHOR, DATE)  
values  
    (?, ?, ?)
```

Hibernate的核心:



Hibernate六大核心接口

- 1、 Configuration接口:负责配置并启动Hibernate
- 2、 SessionFactory接口:负责初始化Hibernate
- 3、 Session接口:负责持久化对象的CRUD操作
- 4、 Transaction接口:负责事务
- 5、 Query接口和Criteria接口:负责执行各种数据库查询

Hello world

- 使用 Hibernate 进行数据持久化操作，通常有如下步骤：
 - 编写持久化类：POJO + 映射文件
 - 获取 Configuration 对象
 - 获取 SessionFactory 对象
 - 获取 Session，打开事务
 - 用面向对象的方式操作数据库
 - 关闭事务，关闭 Session

Configuration 类

- Configuration 类负责管理 Hibernate 的配置信息。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（对应 `hibernate.cfg.xml` 文件）。
 - 持久化类与数据表的映射关系（`*.hbm.xml` 文件）
- 创建 Configuration 的两种方式
 - 属性文件（`hibernate.properties`）：
 - `Configuration cfg = new Configuration();`
 - Xml文件（`hibernate.cfg.xml`）
 - `Configuration cfg = new Configuration().configure();`
 - Configuration 的 `configure` 方法还支持带参数的访问：
 - `File file = new File("simpleit.xml");`
 - `Configuration cfg = new Configuration().configure(file);`

SessionFactory 接口

- 针对单个数据库映射关系经过编译后的内存镜像，是线程安全的。
- SessionFactory 对象一旦构造完毕，即被赋予特定的配置信息
- SessionFactory是生成Session的工厂
- 构造 SessionFactory 很消耗资源，一般情况下一个应用中只初始化一个 SessionFactory 对象。
- Hibernate4 新增了一个 ServiceRegistry 接口，所有基于 Hibernate 的配置或者服务都必须统一向这个 ServiceRegistry 注册后才能生效
- Hibernate4 中创建 SessionFactory 的步骤

```
//1. 创建 Configuration 对象
Configuration configuration = new Configuration().configure();
//2. 创建 ServiceRegistry 对象
ServiceRegistry serviceRegistry =
    new ServiceRegistryBuilder().applySettings(configuration.getProperties()).buildServiceRegistry();
//3. 创建 SessionFactroy 对象
SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);
```

Session 接口

- Session 是应用程序与数据库之间交互操作的一个**单线程对象**，是 Hibernate 运作的中心，所有持久化对象必须在 session 的管理下才可以进行持久化操作。此对象的生命周期很短。Session 对象有一个一级缓存，显式执行 flush 之前，所有的持久层操作的数据都缓存在 session 对象处。**相当于 JDBC 中的 Connection。**

Session 接口

- 持久化类与 Session 关联起来后就具有了持久化的能力。
- Session 类的方法：
 - 取得持久化对象的方法：get() load()
 - 持久化对象都得保存，更新和删除：
save(),update(),saveOrUpdate(),delete()
 - 开启事务: beginTransaction().
 - 管理 Session 的方法：isOpen(),flush(), clear(), evict(), close()等

Transaction(事务)

- 代表一次原子操作，它具有数据库事务的概念。所有持久层都应该在事务管理下进行，即使是只读操作。

Transaction tx = session.beginTransaction();

- 常用方法:
 - commit():提交相关联的session实例
 - rollback():撤销事务操作
 - wasCommitted():检查事务是否提交

Hibernate 配置文件的两个配置项

- hbm2ddl.auto : 该属性可帮助程序员实现正向工程, 即由 java 代码生成数据库脚本, 进而生成具体的表结构. 。取值 create | update | create-drop | validate
 - create : 会根据 .hbm.xml 文件来生成数据表, 但是每次运行都会删除上一次的表 , 重新生成表, 哪怕二次没有任何改变
 - create-drop : 会根据 .hbm.xml 文件生成表, 但是SessionFactory一关闭, 表就自动删除
 - update : **最常用的属性值** , 也会根据 .hbm.xml 文件生成表, 但若 .hbm.xml 文件和数据库中对应的数据表的表结构不同, Hiberante 将更新数据表结构 , 但不会删除已有的行和列
 - validate : 会和数据库中的表进行比较, 若 .hbm.xml 文件中的列在数据表中不存在 , 则抛出异常
- format_sql : 是否将 SQL 转化为格式良好的 SQL . 取值 true | false

通过 Session 操纵对象

Session 概述

- Session 接口是 Hibernate 向应用程序提供的操纵数据库的最主要的接口, 它**提供了基本的保存, 更新, 删除和加载 Java 对象的方法.**
- **Session 具有一个缓存, 位于缓存中的对象称为持久化对象, 它和数据库中的相关记录对应.** Session 能够在某些时间点, 按照缓存中对象的变化来执行相关的 SQL 语句, 来同步更新数据库, 这一过程被称为刷新缓存(flush)
- **站在持久化的角度, Hibernate 把对象分为 4 种状态:** 持久化状态, 临时状态, 游离状态, 删除状态. Session 的特定方法能使对象从一个状态转换到另一个状态.

Session 的 save() 方法

- Session 的 save() 方法使一个临时对象转变为持久化对象
- Session 的 save() 方法完成以下操作：
 - 把 News 对象加入到 Session 缓存中, 使它进入持久化状态
 - 选用映射文件指定的标识符生成器, 为持久化对象分配唯一的 OID.
在使用代理主键的情况下, setId() 方法为 News 对象设置 OID 使无效的.
 - 计划执行一条 insert 语句 : 在 flush 缓存的时候
- Hibernate 通过持久化对象的 OID 来维持它和数据库相关记录的对应关系. 当 News 对象处于持久化状态时, **不允许程序随意修改它的 ID**
- **persist() 和 save() 区别 :**
 - 当对一个 OID 不为 Null 的对象执行 save() 方法时, 会把该对象以一个新的 oid 保存到数据库中; 但执行 persist() 方法时会抛出一个异常.

Employee.java

```
• public class Employee {  
•     private int id;  
•     private String firstName;  
•     private String lastName;  
•     private int salary;  
  
•     public Employee() {}  
•     public Employee(String fname, String lname, int salary) {  
•         this.firstName = fname;  
•         this.lastName = lname;  
•         this.salary = salary;  
•     }  
•     public int getId() {  
•         return id;  
•     }  
•     public void setId( int id ) {  
•         this.id = id;  
•     }  
•     public String getFirstName() {  
•         return firstName;  
•     }  
•     public void setFirstName( String first_name ) {  
•         this.firstName = first_name;  
•     }  
•     public String getLastName() {  
•         return lastName;  
•     }  
•     public void setLastName( String last_name ) {  
•         this.lastName = last_name;  
•     }  
•     public int getSalary() {  
•         return salary;  
•     }  
•     public void setSalary( int salary ) {  
•         this.salary = salary;  
•     }  
• }
```

- create table EMPLOYEE (
 - id INT NOT NULL auto_increment,
 - first_name VARCHAR(20) default NULL,
 - last_name VARCHAR(20) default NULL,
 - salary INT default NULL,
 - PRIMARY KEY (id)
-);

创建映射配置文件

- `<?xml version="1.0" encoding="utf-8"?>`
- `<!DOCTYPE hibernate-mapping PUBLIC`
- `"-//Hibernate/Hibernate Mapping DTD//EN"`
- `"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">`
- `<hibernate-mapping>`
- `<class name="Employee" table="EMPLOYEE">`
- `<meta attribute="class-description">`
- `This class contains the employee detail.`
- `</meta>`
- `<id name="id" type="int" column="id">`
- `<generator class="native"/>`
- `</id>`
- `<property name="firstName" column="first_name" type="string"/>`
- `<property name="lastName" column="last_name" type="string"/>`
- `<property name="salary" column="salary" type="int"/>`
- `</class>`
- `</hibernate-mapping>`

hibernate.cfg.xml

```
• <hibernate-configuration>
•   <session-factory>
•
•       <!-- 配置连接数据库的基本信息 -->
•       <property name="connection.username">root</property>
•       <property name="connection.password">333333</property>
•       <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
•       <property name="connection.url">jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf-
8</property>
•
•       <!-- 配置 hibernate 的基本信息 -->
•       <!-- hibernate 所使用的数据库方言 -->
•       <property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>
•
•       <!-- 执行操作时是否在控制台打印 SQL -->
•       <property name="show_sql">true</property>
•
•       <!-- 是否对 SQL 进行格式化 -->
•       <property name="format_sql">true</property>
•
•       <!-- 指定自动生成数据表的策略 -->
•       <property name="hbm2ddl.auto">update</property>
•
•       <!-- 指定关联的 .hbm.xml 文件 -->
•       <mapping resource="com/atguigu/hibernate/helloworld/News.hbm.xml"/>
•
•       <mapping resource="com/atguigu/hibernate/helloworld/Employee.hbm.xml"/>
•       <mapping resource="com/lihui/hibernate/single_n_n/Item.hbm.xml"/>
•       <mapping resource="com/lihui/hibernate/single_n_n/Category.hbm.xml"/>
•
•   </session-factory>
•
• </hibernate-configuration>
```

- `factory = new Configuration().configure().buildSessionFactory();`
- `Session session = factory.openSession();`
- `Transaction tx = null;`
- `Integer employeeID = null;`
- `try{`
- `tx = session.beginTransaction();`
- `Employee employee = new Employee(fname, lname, salary);`
- `employeeID = (Integer) session.save(employee);`
- `tx.commit();`
- `}catch (HibernateException e) {`
- `if (tx!=null) tx.rollback();`
- `e.printStackTrace();`
- `}finally {`
- `session.close();`
- `}`

Session 的 get() 和 load() 方法

- 都可以根据跟定的 OID 从数据库中加载一个持久化对象
- 区别:
 - 当数据库中不存在与 OID 对应的记录时, load() 方法抛出 ObjectNotFoundException 异常, 而 get() 方法返回 null
 - 两者采用不同的延迟检索策略: load 方法支持延迟加载策略。而 get 不支持。

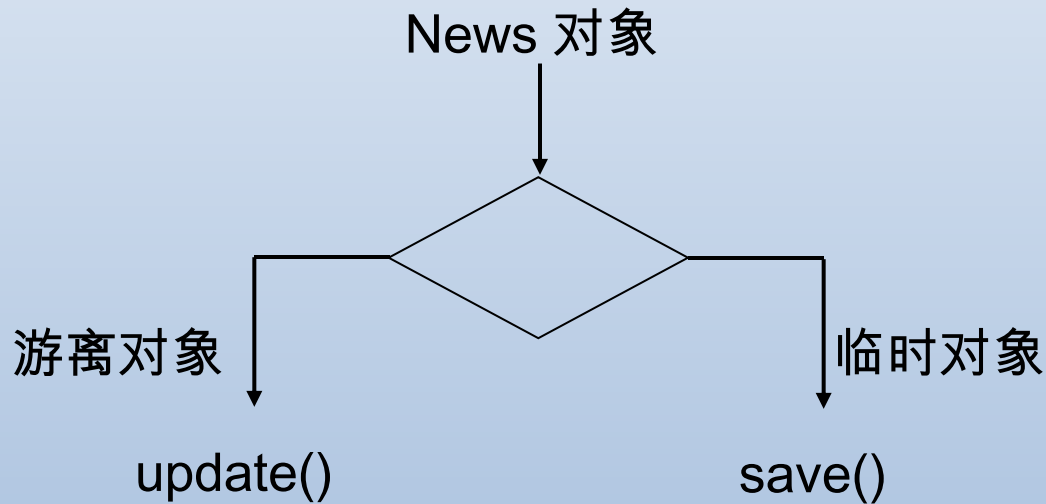
Session 的 update() 方法

- Session 的 update() 方法使一个游离对象转变为持久化对象, 并且计划执行一条 update 语句.
- 若希望 Session 仅当修改了 News 对象的属性时, 才执行 update() 语句, 可以把映射文件中 <class> 元素的 **select-before-update** 设为 true. 该属性的默认值为 false
- **当 update() 方法关联一个游离对象时, 如果在 Session 的缓存中已经存在相同 OID 的持久化对象, 会抛出异常**
- 当 update() 方法关联一个游离对象时, 如果在数据库中不存在相应的记录, 也会抛出异常.

- Session session = factory.openSession();
- Transaction tx = null;
- try{
- tx = session.beginTransaction();
- Employee employee =
- (Employee)session.get(Employee.class, EmployeeID);
- employee.setSalary(salary);
- session.update(employee);
- tx.commit();
- }catch (HibernateException e) {
- if (tx!=null) tx.rollback();
- e.printStackTrace();
- }finally {
- session.close();
- }

Session 的 saveOrUpdate() 方法

- Session 的 saveOrUpdate() 方法同时包含了 save() 与 update() 方法的功能



- 判定对象为临时对象的标准
 - Java 对象的 OID 为 null
 - 映射文件中为 `<id>` 设置了 **unsaved-value** 属性, 并且 Java 对象的 OID 取值与这个 `unsaved-value` 属性值匹配

Session 的 delete() 方法

- Session 的 delete() 方法既可以删除一个游离对象, 也可以删除一个持久化对象
- Session 的 delete() 方法处理过程
 - 计划执行一条 delete 语句
 - 把对象从 Session 缓存中删除, 该对象进入删除状态.
- Hibernate 的 cfg.xml 配置文件中有一个 **hibernate.use_identifier_rollback** 属性, 其默认值为 false, 若把它设为 true, 将改变 delete() 方法的运行行为: delete() 方法会把持久化对象或游离对象的 OID 设置为 null, 使它们变为临时对象

- Session session = factory.openSession();
- Transaction tx = null;
- try{
- tx = session.beginTransaction();
- Employee employee =
- (Employee)session.get(Employee.class, EmployeeID);
- session.delete(employee);
- tx.commit();
- }catch (HibernateException e) {
- if (tx!=null) tx.rollback();
- e.printStackTrace();
- }finally {
- session.close();
- }

Hibernate 的配置文件

Hibernate配置文件

- Hibernate 配置文件主要用于配置数据库连接和 Hibernate 运行时所需的各种属性
- 每个 Hibernate 配置文件对应一个 Configuration 对象
- Hibernate配置文件可以有两种格式:
 - hibernate.properties
 - [hibernate.cfg.xml](#)

hibernate.cfg.xml的常用属性

- JDBC 连接属性

- connection.url : 数据库URL
- connection.username : 数据库用户名
- connection.password : 数据库用户密码
- connection.driver_class : 数据库JDBC驱动
- **dialect** : 配置数据库的方言，根据底层的数据库不同产生不同的 sql 语句，Hibernate 会针对数据库的特性在访问时进行优化

hibernate.cfg.xml的常用属性

- C3P0 数据库连接池属性

- hibernate.c3p0.max_size: 数据库连接池的最大连接数
- hibernate.c3p0.min_size: 数据库连接池的最小连接数
- hibernate.c3p0.timeout: 数据库连接池中连接对象在多长时间没有使用过后，就应该被销毁
- hibernate.c3p0.max_statements: 缓存 Statement 对象的数量
- hibernate.c3p0.idle_test_period: 表示连接池**检测线程**多长时间检测一次池内的所有链接对象是否超时. 连接池本身不会把自己从连接池中移除，而是专门有一个线程按照一定的时间间隔来做这件事，这个线程通过比较连接对象最后一次被使用时间和当前时间的时间差来和 timeout 做对比，进而决定是否销毁这个连接对象。
- hibernate.c3p0.acquire_increment: 当数据库连接池中的连接耗尽时，同一时刻获取多少个数据库连接

hibernate.cfg.xml的常用属性

- 其他

- show_sql : 是否将运行期生成的SQL输出到日志以供调试。取值 true | false
- format_sql : 是否将 SQL 转化为格式良好的 SQL . 取值 true | false
- hbm2ddl.auto : 在启动和停止时自动地创建 , 更新或删除数据库模式。取值 create | update | create-drop | validate
- hibernate.jdbc.fetch_size
- hibernate.jdbc.batch_size

jdbc.fetch_size 和 jdbc.batch_size

- hibernate.jdbc.fetch_size：实质是调用 Statement.setFetchSize() 方法**设定 JDBC 的 Statement 读取数据的时候每次从数据库中取出的记录条数**。
 - 例如一次查询1万条记录，对于Oracle的JDBC驱动来说，是不会 1 次性把1万条取出来的，而只会取出 fetchSize 条数，当结果集遍历完了这些记录以后，再去数据库取 fetchSize 条数据。因此大大节省了无谓的内存消耗。Fetch Size设的越大，读数据库的次数越少，速度越快；Fetch Size越小，读数据库的次数越多，速度越慢。Oracle数据库的JDBC驱动默认的Fetch Size = 10，是一个保守的设定，根据测试，当Fetch Size=50时，性能会提升1倍之多，当 **fetchSize=100**，性能还能继续提升20%，Fetch Size继续增大，性能提升的就不显著了。并不是所有的数据库都支持Fetch Size特性，例如MySQL就不支持
- hibernate.jdbc.batch_size：**设定对数据库进行批量删除，批量更新和批量插入的时候的批次大小**，类似于设置缓冲区大小的意思。batchSize 越大，批量操作时向数据库发送sql的次数越少，速度就越快。
 - 测试结果是当Batch Size=0的时候，使用Hibernate对Oracle数据库删除1万条记录需要25秒，Batch Size = 50的时候，删除仅仅需要5秒！Oracle数据库 **batchSize=30** 的时候比较合适。

对象关系映射文件

POJO 类和数据库的映射文件*.hbm.xml

- POJO 类和关系数据库之间的映射可以用一个XML文档来定义。
- 通过 POJO 类的数据库映射文件，Hibernate可以理解持久化类和数据表之间的对应关系，也可以理解持久化类属性与数据库表列之间的对应关系
- 在运行时 Hibernate 将根据这个映射文件来生成各种 SQL 语句
- 映射文件的扩展名为 .hbm.xml

映射文件说明

- hibernate-mapping
 - 类层次 : class
 - 主键 : id
 - 基本类型:property
 - 实体引用类: many-to-one | one-to-one
 - 集合:set | list | map | array
 - one-to-many
 - many-to-many
 - 子类:subclass | joined-subclass
 - 其它:component | any 等
 - 查询语句:query (用来放置查询语句 , 便于对数据库查询的统一管理和优化)
- 每个Hibernate-mapping中可以同时定义多个类. 但更推荐为每个类都创建一个单独的映射文件

hibernate-mapping

```
⑧ auto-import="true"
⑧ catalog
⑧ default-access="property"
⑧ default-cascade="none"
⑧ default-lazy="true"
⑧ package
⑧ schema
```

- hibernate-mapping 是 hibernate 映射文件的根元素
 - schema: 指定所映射的数据库schema的名称。若指定该属性, 则表明会自动添加该 schema 前缀
 - catalog: 指定所映射的数据库catalog的名称。
 - default-cascade(默认为 none): 设置hibernate默认的级联风格. 若配置 Java 属性, 集合映射时没有指定 cascade 属性, 则 Hibernate 将采用此处指定的级联风格。
 - default-access (默认为 property): 指定 Hibernate 的默认的 属性访问策略。默认值为 property, 即使用 getter, setter 方法来访问属性. 若指定 access, 则 Hibernate 会忽略 getter/setter 方法, 而通过反射访问成员变量。
 - default-lazy(默认为 true): 设置 Hibernat morning的延迟加载策略. 该属性的默认值为 true, 即启用延迟加载策略. 若配置 Java 属性映射, 集合映射时没有指定 lazy 属性, 则 Hibernate 将采用此处指定的延迟加载策略
 - auto-import (默认为 true): 指定是否可以在查询语言中使用非全限定的类名 (仅限于本映射文件中的类) 。
 - package (可选): 指定一个包前缀, 如果在映射文档中没有指定全限定的类名, 就使用这个作为包名。

class

- class 元素用于指定类和表的映射

- **name**:指定该持久化类映射的持久化类的类名
- **table**:指定该持久化类映射的表名, Hibernate 默认以持久化类的类名作为表名
- **dynamic-insert**: 若设置为 true, 表示当保存一个对象时, 会动态生成 insert 语句, insert 语句中仅包含所有取值不为 null 的字段. 默认值为 false
- **dynamic-update**: 若设置为 true, 表示当更新一个对象时, 会动态生成 update 语句, update 语句中仅包含所有取值需要更新的字段. 默认值为 false
- **select-before-update**:设置 Hibernate 在更新某个持久化对象之前是否需要先执行一次查询. 默认值为 false
- **batch-size**:指定根据 OID 来抓取实例时每批抓取的实例数.
- **lazy**: 指定是否使用延迟加载.
- **mutable**: 若设置为 true, 等价于所有的 <property> 元素的 update 属性为 false, 表示整个实例不能被更新. 默认为 true.
- **discriminator-value**: 指定区分不同子类的值. 当使用 <subclass/> 元素来定义持久化类的继承关系时需要使用该属性

```
ⓐ abstract="true"
ⓐ batch-size ●
ⓐ catalog
ⓐ check
ⓐ discriminator-value ●
ⓐ dynamic-insert="false" ●
ⓐ dynamic-update="false" ●
ⓐ entity-name
ⓐ lazy="true" ●
ⓐ mutable="true" ●
ⓐ name ●
ⓐ node
ⓐ optimistic-lock="version"
ⓐ persister
ⓐ polymorphism="implicit"
ⓐ proxy
ⓐ rowid
ⓐ schema
ⓐ select-before-update="false" ●
ⓐ subselect
ⓐ table ●
ⓐ where
```

映射对象标识符

- Hibernate 使用对象标识符(OID) 来建立内存中的对象和数据库表中记录的对应关系. 对象的 OID 和数据表的主键对应. Hibernate 通过标识符生成器来为 **主键赋值**
- Hibernate 推荐在数据表中使用代理主键, 即不具备业务含义的字段. 代理主键通常为整数类型, 因为整数类型比字符串类型要节省更多的数据库空间.
- 在对象-关系映射文件中, <id> 元素用来设置对象标识符. <generator> 子元素用来设定标识符生成器.
- Hibernate 提供了标识符生成器接口: IdentifierGenerator, 并提供了各种内置实现

id

saveOrUpdate

通常情况下, Hibernate 建议为持久化类定义一个标识属性, 用于唯一地标识某个持久化实例, 而标识属性则需要映射到底层数据表的主键。

⑧ access
⑧ column ●
⑧ length
⑧ name ●
⑧ node
⑧ type ●
⑧ unsaved-value ●

- **id** : 设定持久化类的 OID 和表的主键的映射
 - **name**: 标识持久化类 OID 的属性名
 - **column**: 设置标识属性所映射的数据表的列名(主键字段的名字).
 - **unsaved-value**: 若设定了该属性, Hibernate 会通过比较持久化类的 OID 值和该属性值来区分当前持久化类的对象是否为临时对象
 - **type**: 指定 Hibernate 映射类型. Hibernate 映射类型是 Java 类型与 SQL 类型的桥梁. 如果没有为某个属性显式设定映射类型, Hibernate 会运用反射机制先识别出持久化类的特定属性的 Java 类型, 然后自动使用与之对应的默认的 Hibernate 映射类型
 - Java 的基本数据类型和包装类型对应相同的 Hibernate 映射类型. 基本数据类型无法表达 null, 所以对于持久化类的 OID 推荐使用包装类型

几乎所有现代的数据库建模理论都推荐不要使用具有实际意义的物理主键，而是推荐使用没有任何实际意义的逻辑主键。尽量避免使用复杂的物理主键，应考虑为数据库增加一列，作为逻辑主键。表面上看，增加逻辑主键增加了数据冗余，但如果从外键关联的角度看，使用逻辑主键的主从表关联中，从表只需增加一个外键列。如果使用多列作为联合主键，则需要在从表中增加多个外键列，如果有多个从表需要增加外键列，则数据冗余更大。

使用物理主键还会增加数据库维护的复杂度，主从表之间的约束关系隐讳难懂，难于维护。

generator



- generator : 设定持久化类设定标识符生成器
 - class: 指定使用的标识符生成器全限定类名或其缩写名

主键生成策略generator

- Hibernate提供的内置标识符生成器:

标识符生成器	描述
increment	适用于代理主键。由Hibernate自动以递增方式生成。
identity	适用于代理主键。由底层数据库生成标识符。
sequence	适用于代理主键。Hibernate根据底层数据库的序列生成标识符，这要求底层数据库支持序列。
hilo	适用于代理主键。Hibernate分局high/low算法生成标识符。
seqhilo	适用于代理主键。使用一个高/低位算法来高效的生成long, short或者int类型的标识符。
native	适用于代理主键。根据底层数据库对自动生成标识符的方式，自动选择identity、sequence或hilo。
uuid.hex	适用于代理主键。Hibernate采用128位的UUID算法生成标识符。
uuid.string	适用于代理主键。UUID被编码成一个16字符长的字符串。
assigned	适用于自然主键。由Java应用程序负责生成标识符。
foreign	适用于代理主键。使用另外一个相关联的对象的标识符。

提示:

UUID 算法会根据 IP 地址, JVM 的启动时间 (精确到 1/4 秒)、系统时间和一个计数器值 (在 JVM 中唯一) 来生成一个 32 位的字符串, 因为通常 UUID 生成的字符串在一个

increment 标识符生成器

- increment 标识符生成器由 **Hibernate** 以递增的方式为代理主键赋值
- Hibernate 会先读取 NEWS 表中的主键的最大值, 而接下来向 NEWS 表中插入记录时, 就在 $\max(id)$ 的基础上递增, 增量为 1.
- 适用范围:
 - 由于 increment 生存标识符机制不依赖于底层数据库系统, 因此它适合所有的数据库系统
 - 适用于只有单个 Hibernate **应用进程** 访问同一个数据库的场合, **在集群环境下不推荐使用它**
 - OID 必须为 long, int 或 short 类型, 如果把 OID 定义为 byte 类型, 在运行时会抛出异常

identity 标识符生成器

- identity 标识符生成器由底层数据库来负责生成标识符, 它要求底层数据库把主键定义为自动增长字段类型
- 适用范围:
 - 由于 identity 生成标识符的机制依赖于底层数据库系统, 因此, 要求底层数据库系统必须支持自动增长字段类型. 支持自动增长字段类型的数据库包括: DB2, Mysql, MSSQLServer, Sybase 等
 - OID 必须为 long, int 或 short 类型, 如果把 OID 定义为 byte 类型, 在运行时会抛出异常

sequence 标识符生成器

- sequence 标识符生成器利用底层数据库提供的序列来生成标识符.

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

- Hibernate 在持久化一个 News 对象时, 先从底层数据库的 news_seq 序列中获得一个唯一的标识号, 再把它作为主键值
- 适用范围:
 - 由于 sequence 生成标识符的机制依赖于底层数据库系统的序列, 因此, 要求底层数据库系统必须支持序列. 支持序列的数据库包括: DB2, Oracle 等
 - OID 必须为 long, int 或 short 类型, 如果把 OID 定义为 byte 类型, 在运行时抛出异常

hilo 标识符生成器

- hilo 标识符生成器由 Hibernate 按照一种 high/low 算法*生成标识符, 它从数据库的特定表的字段中获取 high 值.

```
<id name="id">
  <generator class="hilo">
    <param name="table">HI_TABLE</param>
    <param name="column">NEXT_VALUE</param>
    <param name="max_lo">10</param>
  </generator>
</id>
```

- Hibernate 在持久化一个 News 对象时, 由 Hibernate 负责生成主键值. **hilo 标识符生成器在生成标识符时, 需要读取并修改 HI_TABLE 表中的 NEXT_VALUE 值.**
- 适用范围:
 - 由于 hilo 生存标识符机制不依赖于底层数据库系统, 因此它适合所有的数据库系统
 - OID 必须为 long, int 或 short 类型, 如果把 OID 定义为 byte 类型, 在运行时会抛出异常

native 标识符生成器

- native 标识符生成器依据底层数据库对自动生成标识符的支持能力, 来选择使用 identity, sequence 或 hilo 标识符生成器.
- 适用范围:
 - 由于 native 能根据底层数据库系统的类型, 自动选择合适的标识符生成器, 因此很适合于跨数据库平台开发
 - OID 必须为 long, int 或 short 类型, 如果把 OID 定义为 byte 类型, 在运行时会抛出异常

Property

② name ●
② access ●
② column ●
② formula
② generated="never"
② index
② insert="true"
② lazy="false" ●
② length
② node
② not-null="true"
② optimistic-lock="true"
② precision
② scale
② type ●
② unique-key
② unique="false" ●
② update="true" ●

- property 元素用于指定类的属性和表的字段的映射
 - name: 指定该持久化类的属性的名字
 - column: 指定与类的属性映射的表的字段名. 如果没有设置该属性, Hibernate 将直接使用类的属性名作为字段名.
 - type: 指定 Hibernate 映射类型. Hibernate 映射类型是 Java 类型与 SQL 类型的桥梁. 如果没有为某个属性显式设定映射类型, Hibernate 会运用反射机制先识别出持久化类的特定属性的 Java 类型, 然后自动使用与之对应的默认的 Hibernate 映射类型.
 - not-null: 若该属性值为 true, 表明不允许为 null, 默认为 false
 - access: 指定 Hibernate 的默认的属性访问策略. 默认值为 property, 即使用 getter, setter 方法来访问属性. 若指定 field, 则 Hibernate 会忽略 getter/setter 方法, 而通过反射访问成员变量
 - unique: 设置是否为该属性所映射的数据列添加唯一约束.

Property

```
@name  
@access  
@column  
@formula ●*  
@generated="never"  
@index ●  
@insert="true"  
@lary="false" ●  
@length ●  
@node  
@not-null="true"  
@optimistic-lock="true"  
@precision  
@scale ●  
@type  
@unique-key  
@unique="false" ●  
@update="true"
```

- **property** 元素用于指定类的属性和表的字段的映射
 - **index**: 指定一个字符串的索引名称. 当系统需要 Hibernate 自动建表时, 用于为该属性所映射的数据列创建索引, 从而加快该数据列的查询.
 - **length**: 指定该属性所映射数据列的字段长度
 - **scale**: 指定该属性所映射数据列的小数位数, 对 double, float, decimal 等类型的数据列有效.
 - **formula**: 设置一个 SQL 表达式, Hibernate 将根据它来计算出派生属性的值.
 - 派生属性: 并不是持久化类的所有属性都直接和表的字段匹配, 持久化类的有些属性的值必须在运行时通过计算才能得出来, 这种属性称为派生属性
- 使用 **formula** 属性时
 - formula="(sql)" 的英文括号不能少
 - Sql 表达式中的列名和表名都应该和数据库对应, 而不是和持久化对象的属性对应
 - 如果需要在 formula 属性中使用参数, 这直接使用 where cur.id=id 形式, 其中 id 就是参数, 和当前持久化对象的 id 属性对应的列的 id 值将作为参数传入.

Java 类型, Hibernate 映射类型及 SQL 类型之间的对应关系

Hibernate映射类型	Java类型	标准SQL类型	大小
integer/int	java.lang.Integer/int	INTEGER	4字节
long	java.lang.Long/long	BIGINT	8字节
short	java.lang.Short/short	SMALLINT	2字节
byte	java.lang.Byte/byte	TINYINT	1字节
float	java.lang.Float/float	FLOAT	4字节
double	java.lang.Double/double	DOUBLE	8字节
big_decimal	java.math.BigDecimal	NUMERIC	
character	java.lang.Character/java.lang.String/char	CHAR(1)	定长字符
string	java.lang.String	VARCHAR	变长字符
boolean/ yes_no/true_false	java.lang.Boolean/Boolean	BIT	布尔类型
date	java.util.Date/java.sql.Date	DATE	日期
timestamp	java.util.Date/java.util.Timestamp	TIMESTAMP	日期
calendar	java.util.Calendar	TIMESTAMP	日期
calendar_date	java.util.Calendar	DATE	日期

Java 类型, Hibernate 映射类型及 SQL 类型之间的对应关系

binary	byte[]	BLOB	BLOB
text	java.lang.String	TEXT	CLOB
serializable	实现java.io.Serializable接口的任意Java类	BLOB	BLOB
clob	java.sql.Clob	CLOB	CLOB
blob	java.sql.Blob	BLOB	BLOB
class	java.lang.Class	VARCHAR	定长字符
locale	java.util.Locale	VARCHAR	定长字符
timezone	java.util.TimeZone	VARCHAR	定长字符
currency	java.util.Currency	VARCHAR	定长字符

Java 时间和日期类型的 Hibernate 映射

- 在 Java 中, 代表时间和日期的类型包括: `java.util.Date` 和 `java.util.Calendar`. 此外, 在 JDBC API 中还提供了 3 个扩展了 `java.util.Date` 类的子类: `java.sql.Date`, `java.sql.Time` 和 `java.sql.Timestamp`, 这三个类分别和标准 SQL 类型中的 `DATE`, `TIME` 和 `TIMESTAMP` 类型对应
- 在标准 SQL 中, `DATE` 类型表示日期, `TIME` 类型表示时间, `TIMESTAMP` 类型表示时间戳, 同时包含日期和时间信息.

映射类型	Java 类型	标准 SQL 类型	描述
date	<code>java.util.Date</code> 或 <code>java.sql.Date</code>	DATE	代表日期: yyyy-MM-dd
time	<code>java.util.Date</code> 或 <code>java.sql.Time</code>	TIME	代表时间: hh:mm:ss
timestamp	<code>java.util.Date</code> 或 <code>java.sql.Timestamp</code>	TIMESTAMP	代表时间和日期: yyyymmddhhmiss
calendar	<code>java.util.Calendar</code>	TIMESTAMP	同上
calendar_date	<code>java.util.Calendar</code>	DATE	代表日期: yyyy-MM-dd

使用 Hibernate 内置映射类型

- 以下情况下必须显式指定 Hibernate 映射类型
 - 一个 Java 类型可能对应多个 Hibernate 映射类型. 例如: 如果持久化类的属性为 `java.util.Date` 类型, 对应的 Hibernate 映射类型可以是 `date`, `time` 或 `timestamp`. 此时必须根据对应的数据表的字段的 SQL 类型, 来确定 Hibernate 映射类型. 如果字段为 `DATE` 类型, 那么 Hibernate 映射类型为 `date`; 如果字段为 `TIME` 类型, 那么 Hibernate 映射类型为 `time`; 如果字段为 `TIMESTAMP` 类型, 那么 Hibernate 映射类型为 `timestamp`.

Employee.java

- public class Employee {
- private int id;
- private String first_name;
- private String last_name;
- private int salary;
- public String getFirst_name() {
- return first_name;
- }
- public void setFirst_name(String first_name) {
- this.first_name = first_name;
- }
- ...
- }

Employee.hbm.xml

- `<?xml version="1.0"?>`
- `<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"`
- `"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">`
- `<hibernate-mapping >`
- `<class name="com.atguigu.hibernate.helloworld.Employee" table="EMPLOYEE">`
- `<meta attribute="class-description">`
- `This class contains the employee detail.`
- `</meta>`
- `<id name="id" type="int" column="id">`
- `<generator class="native"/>`
- `</id>`
- `<property name="first_name" column="first_name" type="string"/>`
- `<property name="last_name" column="last_name" type="string"/>`
- `<property name="salary" column="salary" type="int"/>`
- `</class>`
- `</hibernate-mapping>`

hibernate映射集合属性

- 在hibernate中，持久化对象中不仅仅有基本数据类型与类类型的属性，也有List， Map， Set， collection等集合属性

Set集合操作

- 配置：在Xxx.hbm.xml中<set></set>中配置相关的属性
- 在<set></set>中常用的标签属性与子元素：
- name属性：持久化对象中的set属性的属性名对应
- table属性：新建保存该set集合数据的数据表名
- key:子元素：在数据库总保存set数据的key/id
- element子元素：保存set属性的数据标签，同长都要设置type类型

- <!-- 配置set属： name值为持久化类中的set的属性名， table为保存set的表名 -->
- <set name="students" table="stus">
- <!-- 外键 -->
- <key column="stu_id"></key>
- <!-- set保存对象的数据元素， 必须指明数据的类型且字符串要用小写 -->
- <element column="students" type="string"></element>
- </set>

例子

- public class StudentSet {
-
- private int id;
- private String name;
- private int age;
- private Set<String> hobby;
- ...
- }

StudentSet.hbm.xml

- `<?xml version="1.0" encoding="UTF-8"?>`
- `<!DOCTYPE hibernate-mapping SYSTEM "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd" >`
- `<hibernate-mapping>`
- `<!-- 一个class标签对应一个实体类，name属性指定实体类名称，table属性指定关联的数据库表 -->`
- `<class name="com.joe.entity.StudentSet" table="stu_set_tab">`
- `<!-- 主键 -->`
- `<id name="id" column="stu_id">`
- `<!-- 提供ID自增的策略 native会根据数据库自行判断 -->`
- `<generator class="native"></generator>`
- `</id>`
- `<!-- 其他属性，name对应实体类的属性，column对应关系型数据库表的列 -->`
- `<property name="name" column="stu_name"></property>`
- `<property name="age" column="stu_age"></property>`
- `<!-- 集合属性 -->`
- `<set name="hobby" table="hobby_tab">`
- `<key column="student_id"></key>`
- `<element type="string" column="hobby"></element>`
- `</set>`
- `</class>`
- `</hibernate-mapping>`


```

• public static void add( ){
•
•         SessionFactory sessionFactory;
•         Transaction tx=null;
•
•         Session session=null;
•         try{
•             Configuration cfg = new Configuration().configure();
•             ServiceRegistry sr = new ServiceRegistryBuilder().applySettings(
•                 cfg.getProperties()).buildServiceRegistry();
•             sessionFactory = cfg.buildSessionFactory(sr);
•             session = sessionFactory.openSession();
•
•             tx=session.beginTransaction();
•             StudentSet student=new StudentSet();
•             student.setName("zhangsan555");
•             student.setAge(20);
•
•             @SuppressWarnings({ "rawtypes", "unchecked" })
•             Set<String> set=new HashSet();
•             set.add("basketball555");
•             set.add("swimming555");
•
•             student.setHobby(set);
•
•             session.save(student);
•
•
•             tx.commit();
•         }catch(HibernateException he){
•             if(tx!=null){
•                 tx.rollback();
•             }
•             he.printStackTrace();
•         }finally{
•             session.close();
•         }
•     }

```

List集合操作

- **list**集合操作与**set**的操作基本一致。在Xxx.hbm.xml文件中配置了<list></list>标签
- <list></list>标签常用的属性与子元素：
- **name**属性：持久化对象中的List属性的属性名对应
- **table**属性：新建保存该集合数据的数据表名
- **key**:子元素：在数据库总保存list数据的key/id
- **element**子元素：保存list属性的数据标签，同长都要设置**type**类型
- **list-index**:子元素：保存在list的属性保存数据的下标索引
- Xxx.hbm.xml文件配置List标签实例

- <!-- 配置list属性。name为持久化对象的属性名
table为保存该属性数据的表名 -->
- <list name="hobbys" table="hobbys_list">
- <key column="hobbys_id"></key>
- <!-- 索引 -->
- <list-index column="position"></list-index>
- <!-- 报list属性的数据字段 需要制定类型 -->
- <element type="string"
column="hobbys"></element>
- </list>

Map集合的操作

- `map`属性使用也没有多大的差别，在`Xxx.hbm.xml`中，使用`<map></map>`标签配置
- `<map></map>`的常用属性与子元素：
- `name`属性：持久化对象中的`map`属性的属性名对应
- `table`属性：新建保存该`map`集合数据的数据表名
- `key`:子元素：在数据库总保存`set`数据的`key/id`
- `element`子元素：保存`map`属性的数据标签，同长都要设置`type`类型
- `map-key`子元素：`map`中保存数据的`key`

Xxx.hbm.xml配置实例

- `<map name="hobby" table="hobby_map">`
- `<key column="name"></key>`
- `<map-key column="hobby_mapkey"`
 `type="string"></map-key>`
- `<element column="hobby"`
 `type="string"></element>`
- `</map>`

注释

- 你已经看到 **Hibernate** 如何使用 **XML** 映射文件来完成从 **POJO** 到数据库表的数据转换的，反之亦然。**Hibernate** 注释是无需使用 **XML** 文件来定义映射的最新方法。你可以额外使用注释或直接代替 **XML** 映射元数据。
- **Hibernate** 注释是一种强大的来给对象和关系映射表提供元数据的方法。所有的元数据被添加到 **POJO java** 文件代码中，这有利于用户在开发时更好的理解表的结构和 **POJO**。

- create table EMPLOYEE (
- id INT NOT NULL auto_increment,
- first_name VARCHAR(20) default NULL,
- last_name VARCHAR(20) default NULL,
- salary INT default NULL,
- PRIMARY KEY (id)
-);

用带有注释的 Employee 类来映射使用定义好的 Employee 表的对象

```
• import javax.persistence.*;

• @Entity
• @Table(name = "EMPLOYEE")
• public class Employee {
•     @Id @GeneratedValue
•     @Column(name = "id")
•     private int id;

•     @Column(name = "first_name")
•     private String firstName;

•     @Column(name = "last_name")
•     private String lastName;

•     @Column(name = "salary")
•     private int salary;

•     public Employee() {}
•     public int getId() {
•         return id;
•     }
•     public void setId( int id ) {
•         this.id = id;
•     }
•     public String getFirstName() {
•         return firstName;
•     }
•     public void setFirstName( String first_name ) {
•         this.firstName = first_name;
•     }
•     public String getLastName() {
•         return lastName;
•     }
•     public void setLastName( String last_name ) {
•         this.lastName = last_name;
•     }
•     public int getSalary() {
•         return salary;
•     }
•     public void setSalary( int salary ) {
•         this.salary = salary;
•     }
• }
```


- **Hibernate** 检测到 **@Id** 注释字段并且认定它应该在运行时通过字段直接访问一个对象上的属性。如果你将 **@Id** 注释放在 **getId()** 方法中，你可以通过默认的 **getter** 和 **setter** 方法来访问属性。因此，所有其它注释也放在字段或是 **getter** 方法中，决定于选择的策略。

- **@Entity** 注释
- EJB 3 标准的注释包含在 `javax.persistence` 包，所以我们第一步需要导入这个包。第二步我们对 `Employee` 类使用 **@Entity** 注释，标志着这个类为一个实体 `bean`，所以它必须含有一个没有参数的构造函数并且在可保护范围是可见的。
- **@Table** 注释
- **@table** 注释允许您明确表的详细信息保证实体在数据库中持续存在。
- **@table** 注释提供了四个属性，允许您覆盖的表的名称，目录及其模式,在表中可以对列制定独特的约束。现在我们使用的是表名为 `EMPLOYEE`。
- **@Id** 和 **@GeneratedValue** 注释
- 每一个实体 `bean` 都有一个主键，你在类中可以用 **@Id** 来进行注释。主键可以是一个字段或者是多个字段的组合，这取决于你的表的结构。
- 默认情况下，**@Id** 注释将自动确定最合适的主键生成策略，但是您可以通过使用 **@GeneratedValue** 注释来覆盖掉它。**strategy** 和 **generator** 这两个参数我不打算在这里讨论，所以我们只使用默认键生成策略。让 `Hibernate` 确定使用哪些生成器类型来使代码移植于不同的数据库之间。
- **@Column Annotation**
- **@Column** 注释用于指定某一列与某一个字段或是属性映射的细节信息。您可以使用下列注释的最常用的属性：
 - **name** 属性允许显式地指定列的名称。
 - **length** 属性为用于映射一个值，特别为一个字符串值的列的大小。
 - **nullable** 属性允许当生成模式时，一个列可以被标记为非空。
 - **unique** 属性允许列中只能含有唯一的内容

查询语言

- **Hibernate 查询语言（HQL）**是一种面向对象的查询语言，类似于 **SQL**，但不是去对表和列进行操作，而是面向对象和它们的属性。**HQL** 查询被 **Hibernate** 翻译为传统的 **SQL** 查询从而对数据库进行操作

执行HQL查询的步骤

- 1、获得Hibernate Session对象
- 2、编写HQL语句
- 3、调用Session的createQuery方法创建查询对象
- 4、如果HQL语句包含参数，则调用Query的setXxx方法为参数赋值
- 5、调用Query对象的list等方法返回查询结果。

例子

- ```
private void query(){
```
- ```
    Session session = HibernateUtil.getSession();
```
- ```
 Transaction tx = session.beginTransaction();
```
- ```
    //以HQL语句创建Query对象，执行setString方法为HQL语句的参数赋值
```
- ```
 //Query调用list方法访问查询的全部实例
```
- ```
    List list = session.createQuery("select distinct p from Person p where
```
- ```
 p.name=:name")
```
- ```
                                .setString("name", "chenssy").list();
```
- ```
 //遍历查询结果
```
- ```
    for (Iterator iterator = list.iterator();iterator.hasNext();) {
```
- ```
 Person p = (Person) iterator.next();
```
- ```
        System.out.println("id="+p.getId()+"age="+p.getAge());
```
- ```
 }
```
- ```
    session.close();
```
- ```
}
```

# FROM 语句

- 在存储中加载一个完整并持久的对象
- `String hql = "FROM Employee";`
- `Query query = session.createQuery(hql);`
- `List results = query.list();`

# SELECT 语句

- SELECT 语句比 from 语句提供了更多的对结果集的控制。如果你只想得到对象的几个属性而不是整个对象你需要使用 SELECT 语句
- `String hql = "SELECT E.firstName FROM Employee E";`
- `Query query = session.createQuery(hql);`
- `List results = query.list();`

# WHERE 语句

- 如果你想要精确地从数据库存储中返回特定对象，你需要使用 WHERE 语句
- `String hql = "FROM Employee E WHERE E.id = 10";`
- `Query query = session.createQuery(hql);`
- `List results = query.list();`



# ORDER BY 语句

- 为了给 HSQ 查询结果进行排序，你将需要使用 ORDER BY 语句。你能利用任意一个属性给你的结果进行排序，包括升序或降序排序。
- `String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";`
- `Query query = session.createQuery(hql);`
- `List results = query.list();`

# GROUP BY 语句

- 这一语句允许 Hibernate 将信息从数据库中提取出来，并且基于某种属性的值将信息进行编组,通常而言,该语句会使用得到的结果来包含一个聚合值
- `String hql = "SELECT SUM(E.salary), E.firstName  
FROM Employee E " +`
- `"GROUP BY E.firstName";`
- `Query query = session.createQuery(hql);`
- `List results = query.list();`

# 使用命名参数

- Hibernate 的 HQL 查询功能支持命名参数。这使得 HQL 查询功能既能接受来自用户的简单输入，又无需防御 SQL 注入攻击。
- `String hql = "FROM Employee E WHERE E.id = :employee_id";`
- `Query query = session.createQuery(hql);`
- `query.setParameter("employee_id",10);`
- `List results = query.list();`

# UPDATE 语句

- UPDATE 语句能够更新一个或多个对象的一个或多个属性
- `String hql = "UPDATE Employee set salary = :salary"`  
`" +`
- `"WHERE id = :employee_id";`
- `Query query = session.createQuery(hql);`
- `query.setParameter("salary", 1000);`
- `query.setParameter("employee_id", 10);`
- `int result = query.executeUpdate();`
- `System.out.println("Rows affected: " + result);`

# DELETE 语句

- DELETE 语句可以用来删除一个或多个对象
- `String hql = "DELETE FROM Employee " +`
- `"WHERE id = :employee_id";`
- `Query query = session.createQuery(hql);`
- `query.setParameter("employee_id", 10);`
- `int result = query.executeUpdate();`
- `System.out.println("Rows affected: " + result);`

# INSERT 语句

- HQL 只有当记录从一个对象插入到另一个对象时才支持 INSERT INTO 语句。下面是使用 INSERT INTO 语句的简单的语法:
- `String hql = "INSERT INTO Employee(firstName, lastName, salary)" +`
- `"SELECT firstName, lastName, salary FROM old_employee";`
- `Query query = session.createQuery(hql);`
- `int result = query.executeUpdate();`
- `System.out.println("Rows affected: " + result);`

# 标准查询

- **Hibernate** 提供了操纵对象和相应的 **RDBMS** 表中可用的数据的替代方法。一种方法是标准的 **API**，它允许你建立一个标准的可编程查询对象来应用过滤规则和逻辑条件。
- **Hibernate Session** 接口提供了 **createCriteria()** 方法，可用于创建一个 **Criteria** 对象，使当您的应用程序执行一个标准查询时返回一个持久化对象的类的实例。

# 标准查询

- Criteria cr =  
session.createCriteria(Employee.class);
- List results = cr.list();



# 对标准的限制

- 你可以使用 **Criteria** 对象可用的 **add()** 方法去添加一个标准查询的限制
- 以下是一个示例，它实现了添加一个限制，令返回工资等于 2000 的记录：
- ```
Criteria cr =  
    session.createCriteria(Employee.class);
```
- ```
cr.add(Restrictions.eq("salary", 2000));
```
- ```
List results = cr.list();
```

Employee.java

```
• public class Employee {  
•     private int id;  
•     private String firstName;  
•     private String lastName;  
•     private int salary;  
  
•     public Employee() {}  
•     public Employee(String fname, String lname, int salary) {  
•         this.firstName = fname;  
•         this.lastName = lname;  
•         this.salary = salary;  
•     }  
•     public int getId() {  
•         return id;  
•     }  
•     public void setId( int id ) {  
•         this.id = id;  
•     }  
•     public String getFirstName() {  
•         return firstName;  
•     }  
•     public void setFirstName( String first_name ) {  
•         this.firstName = first_name;  
•     }  
•     public String getLastName() {  
•         return lastName;  
•     }  
•     public void setLastName( String last_name ) {  
•         this.lastName = last_name;  
•     }  
•     public int getSalary() {  
•         return salary;  
•     }  
•     public void setSalary( int salary ) {  
•         this.salary = salary;  
•     }  
• }
```

- create table EMPLOYEE (
 - id INT NOT NULL auto_increment,
 - first_name VARCHAR(20) default NULL,
 - last_name VARCHAR(20) default NULL,
 - salary INT default NULL,
 - PRIMARY KEY (id)
-);

映射文件

- `<?xml version="1.0" encoding="utf-8"?>`
- `<!DOCTYPE hibernate-mapping PUBLIC`
- `"-//Hibernate/Hibernate Mapping DTD//EN"`
- `"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">`
- `<hibernate-mapping>`
- `<class name="Employee" table="EMPLOYEE">`
- `<meta attribute="class-description">`
- `This class contains the employee detail.`
- `</meta>`
- `<id name="id" type="int" column="id">`
- `<generator class="native"/>`
- `</id>`
- `<property name="firstName" column="first_name" type="string"/>`
- `<property name="lastName" column="last_name" type="string"/>`
- `<property name="salary" column="salary" type="int"/>`
- `</class>`
- `</hibernate-mapping>`

JUnit进行测试

JUnit4.0的基本使用，主要是几个注解的使用方法：

@Before: 用在方法上面，表示这个方法会在每一个测试方法之前运行，会运行多次，一般用于测试方法的一些初始化操作。

@After: 用在方法上面，表示这个方法会在每一个测试方法之前运行，会运行多次，一般用于测试方法资源的释放。

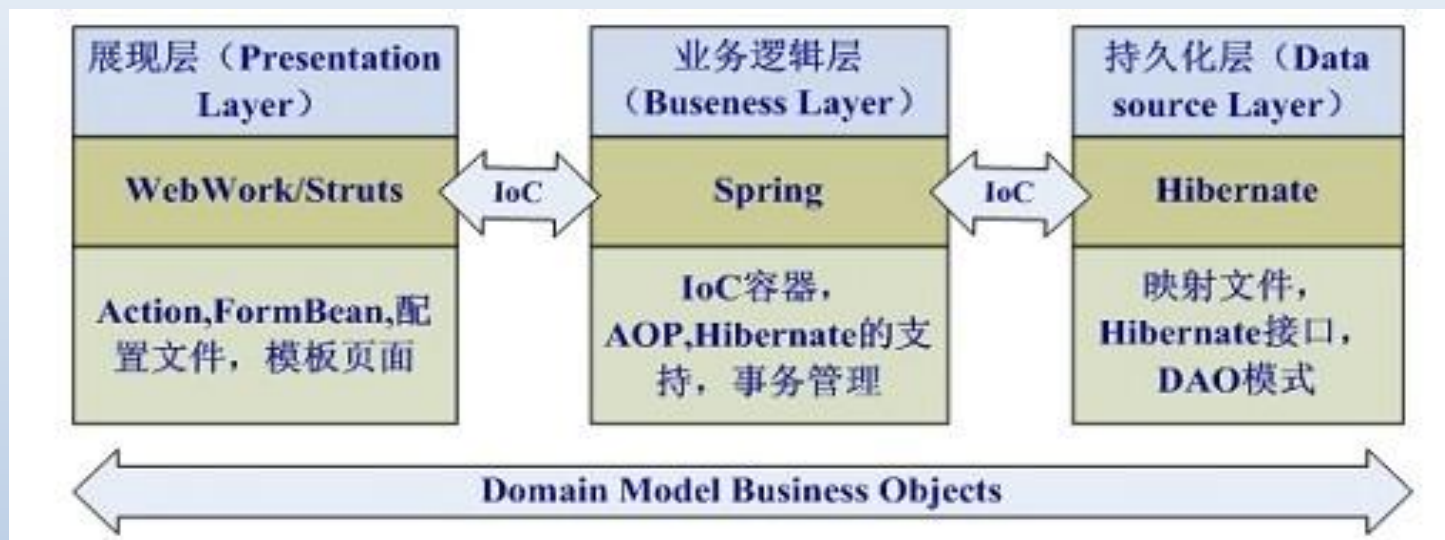
@Test: 用在方法上面，表示这个方法就是测试方法，在这里可以测试期望异常和超时时间。

@Ignore: 用在方法上面，表示这个方法是忽略的测试方法。

@BeforeClass: 只能用在静态方法上面，表示针对整个类中所有测试方法只执行一次，在**@Before**方法之前运行。一般用于类的一些初始化的操作。

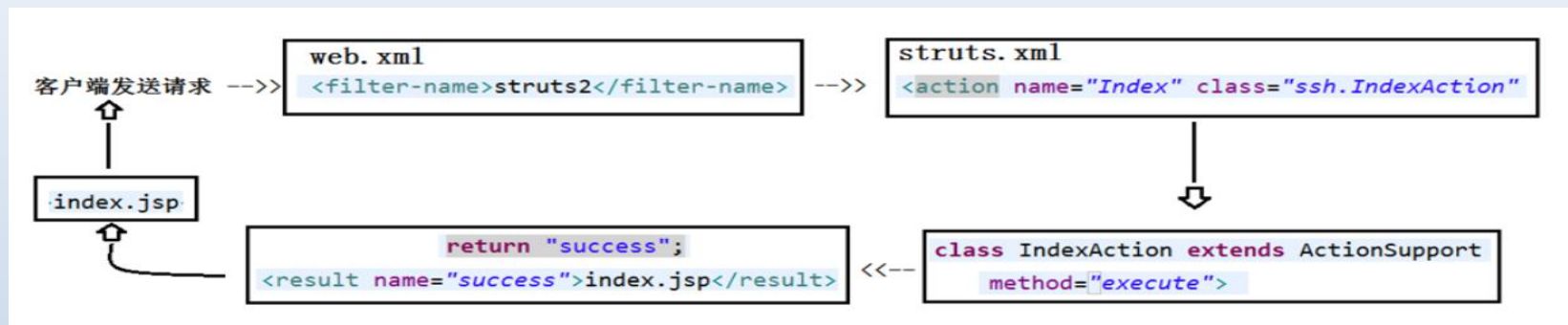
@AfterClass: 只能用在静态方法上面，表示针对整个类中所有测试方法只执行一次，在**@After**方法之后运行。一般用于类的一些资源释放的操作。

Struts、Spring、Hibernate 的整合



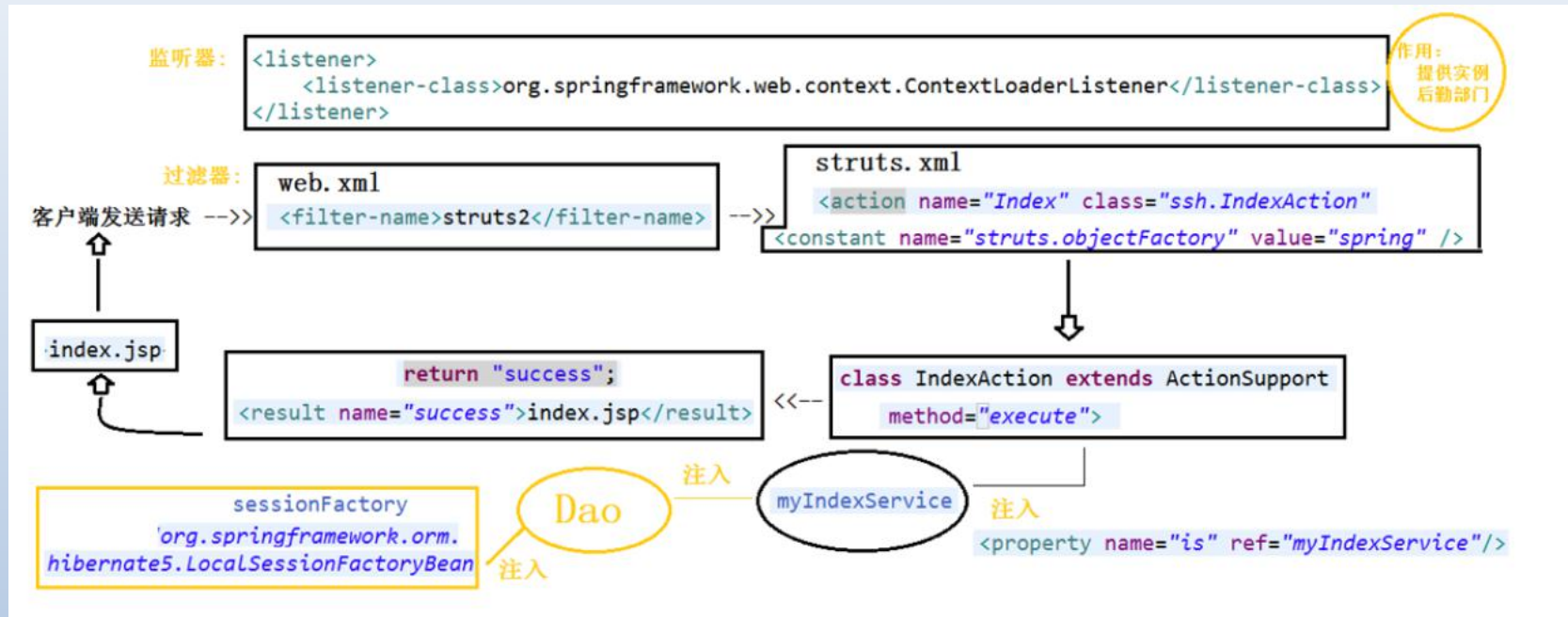
- 从图中可以看出，前端使用 Struts 等框架完成，后端采用 Hibernate 访问数据库。而 Spring 主要运行在 Struts 和 Hibernate 的中间，一般情况下，Spring 负责降低 Web 层和数据库层之间的耦合性，或者说，让 Struts 中的 Action 在调用 Hibernate 中的 DAO 时，尽量降低耦合性

例子struts2框架的运行结构



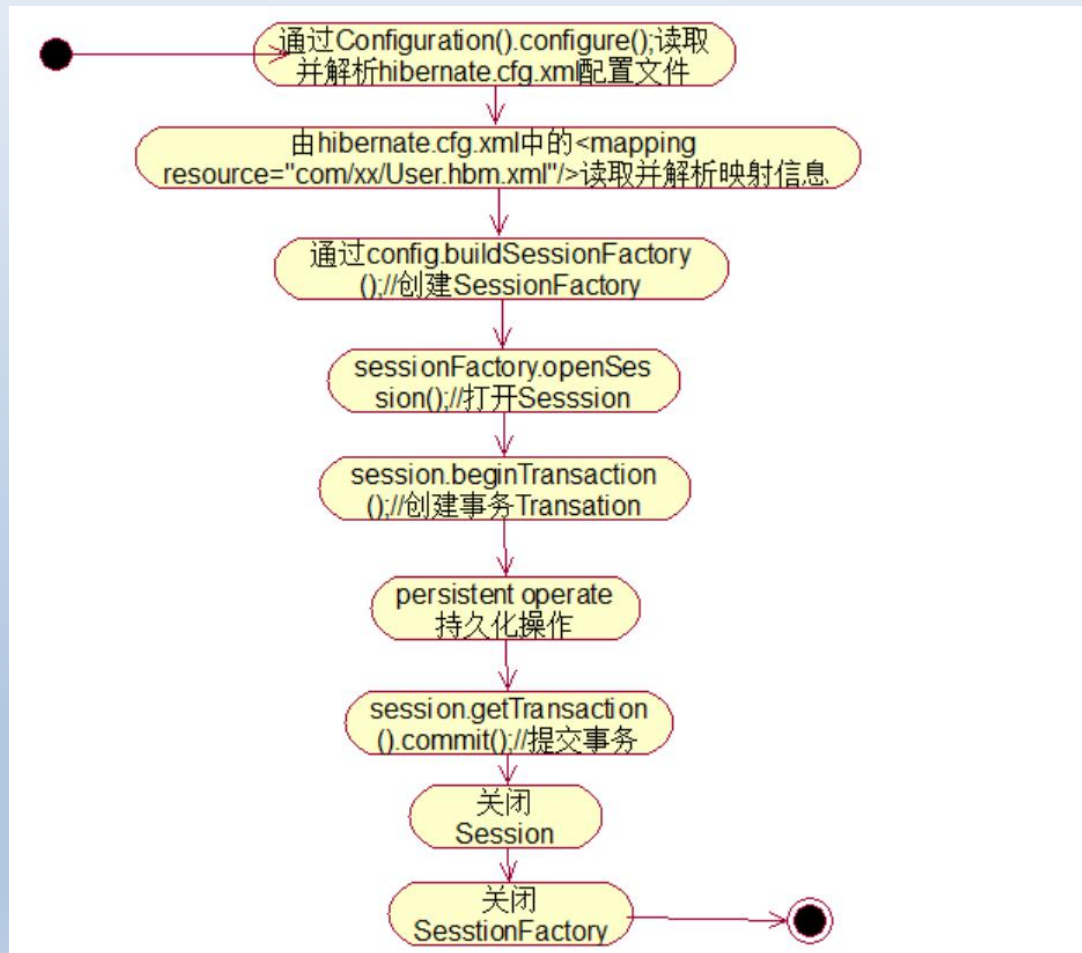
客户端发送请求(HttpServletRequest)到服务器，服务器接收到请求就先进入web.xml配置文件看看有没有配置过滤器，发现有struts2的过滤器，然后就找到struts.xml配置文件，struts.xml配置文件里有定义一个action，然后就去找到类名叫IndexAction这个类(此action类必须是继承ActionSupport接口)，并且实现了execute()方法，返回一个字符串为"success"给struts.xml配置文件，struts.xml配置文件的action会默认调用IndexAction类的execute()方法，result接收到了返回的字符串，然后查找结果字符串对应的(Result)，result就会调用你指定的jsp页面将结果呈现，最后响应回给客户端。

spring的流程图:



上图是在struts结构图的基础上加入了spring流程图，在web.xml配置文件中加入了spring的监听器，在struts.xml配置文件中添加“<constant name="struts.objectFactory" value="spring" />”是告知Struts2运行时使用Spring来创建对象，spring在其中主要做的就是注入实例，将所有需要类的实例都由spring管理。

hibernate的核心构成和执行流程图



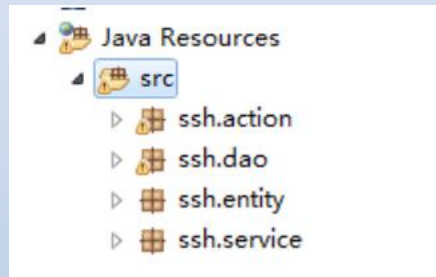
第一步

- 将struts2、spring、hibernate的框架的jar加入类路径
- 以及数据库的jar包加入类路径

第二步在配置文件web.xml配置一个struts2的过滤器和spring监听器

- `<?xml version="1.0" encoding="UTF-8"?>`
- `<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID" version="3.1">`
- `<display-name>ssh</display-name>`
- `<welcome-file-list>`
- `<welcome-file>index.action</welcome-file>`
- `</welcome-file-list>`
- `<!-- struts2的过滤器 -->`
- `<filter>`
- `<filter-name>struts2</filter-name>`
- `<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>`
- `</filter>`
- `<filter-mapping>`
- `<filter-name>struts2</filter-name>`
- `<url-pattern>/*</url-pattern>`
- `</filter-mapping>`
- `<!-- spring的监听器配置开始 -->`
- `<context-param>`
- `<param-name>contextConfigLocation</param-name>`
- `<param-value>classpath:applicationContext.xml</param-value>`
- `</context-param>`
- `<listener>`
- `<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>`
- `</listener>`
- `</web-app>`

第三步在Java Resources下的src目录下创建四个包(package)进行分层



第四步：根据数据库表的字段编写BookCard(实体类)和BookCard.hbm.xml (映射文件)放到ssh.entity包里

```
• public class BookCard {
•     private int cid ;
•     private String name;
•     private String sex ;
•     private Date cardDate;
•     private BigDecimal deposit;
•
•     //定义get()、 set()方法
•     public int getCid() {
•         return cid;
•     }
•     public void setCid(int cid) {
•         this.cid = cid;
•     }
•     public String getName() {
•         return name;
•     }
•     public void setName(String name) {
•         this.name = name;
•     }
•     public String getSex() {
•         return sex;
•     }
•     public void setSex(String sex) {
•         this.sex = sex;
•     }
•     public Date getCardDate() {
•         return cardDate;
•     }
•     public void setCardDate(Date cardDate) {
•         this.cardDate = cardDate;
•     }
•     public BigDecimal getDeposit() {
•         return deposit;
•     }
•     public void setDeposit(BigDecimal deposit) {
•         this.deposit = deposit;
•     }
• }
```

```
• <?xml version="1.0" encoding="UTF-8"?>
• <!DOCTYPE hibernate-mapping PUBLIC
•     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
•     "http://www.hibernate.org/dtd/hibernate-
•         mapping-3.0.dtd">
•
•     <hibernate-mapping>
•         <class name="ssh.entity.BookCard"
•             table="BookCard">
•
•             <!-- 卡号 -->
•             <id name="cid" column="cid">
•                 <generator class="native"></generator>
•             </id>
•             <!-- 姓名 -->
•             <property name="name"
•                 column="name"></property>
•             <!-- 性别 -->
•             <property name="sex"
•                 column="sex"></property>
•             <!-- 办卡日期 -->
•             <property name="cardDate"
•                 column="cardDate"></property>
•             <!-- 押金 -->
•             <property name="deposit"
•                 column="deposit"></property>
•         </class>
•     </hibernate-mapping>
```

第五步：在ssh.service包里编写IndexService(接口类)和IndexServiceImpl(实现类)。

- package ssh.service;
 - import java.util.List;
 - import ssh.dao.IndexDao;
 - import ssh.entity.BookCard;
 - //创建一个IndexService接口类
 - public interface IndexService {
 -
 -
 - public List<BookCard> getAllBookCard();
 - }
- package ssh.service;
 - import java.util.List;
 - import ssh.dao.IndexDao;
 - import ssh.entity.BookCard;
 - //创建IndexServiceImpl(实现类)实现IndexService接口
 - public class IndexServiceImpl implements IndexService {
 -
 - //dao实例使用注入方式
 - private IndexDao id;
 - //用于注入使用
 - public void setId(IndexDao id) {
 - this.id = id;
 - }
 -
 - @Override
 - public List<BookCard> getAllBookCard() {
 - //本类应该编写业务逻辑的代码，
 - //但本例没有业务逻辑，就不用写。
 -
 - //访问数据库的代码，不会出现在service这一层
 - //交给dao来操作数据库
 - List<BookCard> myBookCardList = id.getAllBookCard();
 -
 - //进行其它的业务逻辑操作，比如增加多一个选项，是否过期
 - //本例不需要
 - //....
 -
 - return myBookCardList;
 - }
 - }

第六步：在ssh.dao包里编写IndexDao(接口类)和IndexDaoImpl(实现类)。

- package ssh.dao;
- import java.util.List;
- import ssh.entity.BookCard;
- //创建IndexDao(接口类)
- public interface IndexDao {
-
- public List<BookCard>
- getAllBookCard();
-
- }

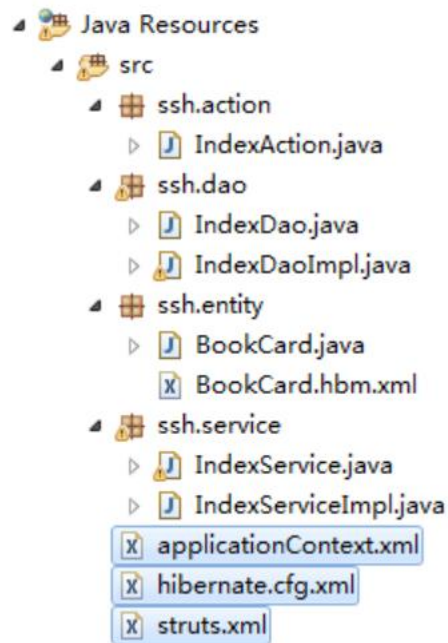
```
• package ssh.dao;
•
• import java.util.List;
•
• import org.hibernate.Session;
• import org.hibernate.SessionFactory;
• import org.hibernate.Query;
•
• import ssh.entity.BookCard;
•
• //创建IndexDaoImpl(实现类)实现IndexDao接口
• public class IndexDaoImpl implements IndexDao {
•     //在SSH的设计理念：要使用某个实例，那么就定义声明一个对象，然后
•     //给它添加set方法（用于spring注入进来）
•     //实现不要关注这个实例来自于那里，以及怎么创建，或者它是谁
•     private SessionFactory sessionFactory;
•     public void setSessionFactory(SessionFactory sessionFactory) {
•         this.sessionFactory = sessionFactory;
•     } @Override
•     public List<BookCard> getAllBookCard() {
•
•         //sessionFactory这个实例可以自己按常规的hibernate传统写法创建
•         //也可以交给spring去托管
•         /*
•         Configuration cfg = new Configuration().configure();
•         sessionFactory = cfg.buildSessionFactory();*/
•
•         //获取session
•         Session session = sessionFactory.openSession();
•
•         //后面当使用JPA的时候，EntityManager 类似于 Session
•         Query query = session.createQuery("from BookCard");
•
•         //将所有数据查询出来并放到List集合里
•         List<BookCard> list = query.list();
•
•         //将集合遍历循环
•         for(BookCard bookCard:list){
•             //打印输出到控制台
•             System.out.println(bookCard);
•         }
•
•         //关闭session
```

第七步：编写IndexAction(action类)

```
• package ssh.action;
• import java.text.DecimalFormat;
• import java.util.List;import com.opensymphony.xwork2.ActionContext;
• import com.opensymphony.xwork2.ActionSupport;import ssh.entity.BookCard;
• import ssh.service.IndexService;
• //创建IndexAction(action类)继承ActionSupport接口
• public class IndexAction extends ActionSupport {
•     private static final long serialVersionUID = 1L;
•     //声明service，但不给它创建具体的实现类的实例，
•     private IndexService is = null;
•     //添加set()方法
•     public void setis(IndexService is) {
•         this.is = is;
•     }
•     //编写execute()方法
•     public String execute() {
•         //获取IndexService实例，调用getAllBookCard()方法
•         //将结果保存到List集合里
•         List<BookCard> myBookCardList = is.getAllBookCard();
•
•         //将查询出来的结构集打印到控制台
•         System.out.println("结果集: "+myBookCardList.size());
•
•         //获取Context上下文对象
•         ActionContext ac = ActionContext.getContext();
•         //将myBookCardList集合添加到上下文对象里
•         ac.put("myBookCardList", myBookCardList);
•
•         //返回一个字符串
•         return "success";
•     }
•
•     //金额格式转换
•     public String formatDouble(double s){
•         DecimalFormat fmat=new DecimalFormat("\u00A4##.0");
•         return fmat.format(s);
•     }
• }
```


第八步：编写struts.xml(struts配置文件)、applicationContext.xml(spring配置文件)、hibernate.cfg.xml(hibernate配置文件)

注：将这些配置文件放到src里。



struts.xml

- `<?xml version="1.0" encoding="UTF-8"?>`
- `<!DOCTYPE struts PUBLIC`
- `"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"`
- `"http://struts.apache.org/dtds/struts-2.3.dtd">`
- `<!-- 上面的头，注意版本，从样例里复制过来 showcase.war\WEB-INF\src\java\struts.xml -->`
- `<struts>`
- `<!-- 告知Struts2运行时使用Spring来创建对象 -->`
- `<constant name="struts.objectFactory" value="spring" />`
- `<!-- 第1步：先定义一个包 -->`
- `<package name="mypck001" extends="struts-default">`
- `<!-- 第2步：定义一个action，配置跳转信息 name 类似于Servlet @WebServlet("/IndexServlet")`
- `http://xxx/xxx/Index.action http://xxx/xxx/Index class 对应于自己写的Action类 当不写method属性时，默认调用的是execute`
- `class="ssh.action.IndexAction" ** new ssh.action.IndexAction()`
- `设计思想：关心了具体的实现类必须改为不要关注那个实现类 加入spring后，struts的action节点的class属性意义发生变化，直接引用spring帮忙创建的实例`
- `-->`
- `<action name="Index" class="myIndexAction">`
- `<!-- 跳转是forward/WEB-INF/是防止jsp不经过action就可以访问-->`
- `<!-- result接收返回的字符串，然后做对应的事情 -->`
- `<result name="success">/WEB-INF/jsp/index.jsp</result>`
- `</action>`
- `</package>`
- `</struts>`

applicationContext.xml

- <?xml version="1.0" encoding="UTF-8"?>
- <beans xmlns="http://www.springframework.org/schema/beans"
- xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
- xmlns:p="http://www.springframework.org/schema/p"
- xmlns:aop="http://www.springframework.org/schema/aop"
- xmlns:context="http://www.springframework.org/schema/context"
- xmlns:jee="http://www.springframework.org/schema/jee"
- xmlns:tx="http://www.springframework.org/schema/tx"
- xsi:schemaLocation="
- http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.2.xsd
- http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
- http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd
- http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-4.2.xsd
- http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.2.xsd">
- <!-- 类似于财务部门一样，类就是钱，所有需要类的实例都由spring去管理 -->
- <bean id="myIndexAction" class="ssh.action.IndexAction" scope="prototype">
- <!-- setIs(myIndexService) -->
- <property name="is" ref="myIndexService"/>
- </bean>
-
- <!-- myIndexService = new ssh.service.IndexServiceImpl() -->
- <bean id="myIndexService" class="ssh.service.IndexServiceImpl" scope="prototype">
- <property name="id" ref="myIndexDao"/>
- </bean>
-
- <bean id="myIndexDao" class="ssh.dao.IndexDaoImpl" scope="prototype">
- <!-- 把sessionFactory 注入给IndexDao -->
- <property name="sessionFactory" ref="sessionFactory" />
- </bean>
-
- <!-- 添加sessionFactory bean，注意，该类是Spring提供的 -->
- <bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean" scope="prototype">
- <!-- 注入Hibernate 配置文件路径,前面要加上 classpath:-->
- <property name="configLocation" value="classpath:hibernate.cfg.xml"/>
- </bean>
-
- </beans>

hibernate.cfg.xml

- `<?xml version="1.0" encoding="utf-8"?>`
- `<!DOCTYPE hibernate-configuration PUBLIC`
- `"-//Hibernate/Hibernate Configuration DTD 3.0//EN"`
- `"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">`
- `<hibernate-configuration>`
- `<session-factory>`
- `<!-- 数据库连接配置 -->`
- `<property name="connection.username">root</property>`
- `<property name="connection.password">333333</property>`
- `<property name="connection.driver_class">com.mysql.jdbc.Driver</property>`
- `<property name="connection.url">jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf-8</property>`
- `<property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>`
- `<!-- 设置默认的数据库连接池 -->`
- `<property name="connection.pool_size">5</property>`
- `<!-- 显示SQL -->`
- `<property name="show_sql">true</property>`
- `<!-- 格式化SQL -->`
- `<property name="format_sql">true</property>`
- `<!-- 根据schema更新数据表的工具 -->`
- `<property name="hbm2ddl.auto">update</property>`
- `<!-- 数据表映射配置文件 -->`
- `<mapping resource="ssh/entity/BookCard.hbm.xml"/>`
- `</session-factory>`
- `</hibernate-configuration>`

第九步：创建一个index.jsp页面将所有数据取出来显示到页面上

```
• <%@ page language="java" contentType="text/html; charset=UTF-8"
•     pageEncoding="UTF-8"%>
• <%@ taglib uri="/struts-tags" prefix="s" %>
• <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
• <html>
• <head>
• <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
• <title>Insert title here</title>
• </head>
• <body>
```

```
• <table border="1">
•     <tr>
•         <td>卡号</td>
•         <td>姓名</td>
•         <td>性别</td>
•         <td>办卡日期</td>
•         <td>押金</td>
•     </tr>
•     <!-- 使用struts2标签库中的iterator将所有数据遍历循环显示出来
•     <s:iterator value="#myBookCardList" status="bcs">
•         <tr>
•             <td><s:property value="cid"></s:property></td>
•             <td><s:property value="name"></s:property></td>
•             <td><s:property value="sex"></s:property></td>
•             <td><s:date name="cardDate" format="yyyy年MM月dd日"></s:date></td>
•             <td><s:property value="%{formatDouble(deposit)}"></s:property></td>
•         </tr>
•     </s:iterator>
•     <!-- 判断查询出来等于0，就显示“没有查找到数据” -->
•     <s:if test="myBookCardList.size()==0">
•         <tr>
•             <td colspan="7">没有查找到数据</td>
•         </tr>
•     </s:if>
• </table>
• </body>
• </html>
```

注：跳转是forward，将jsp放到/WEB-INF/是防止jsp不经过action就可以访问。





sshtest

映射一对多关联关系

- 单向关系：只需单向访问关联端。例如，只能通过老师访问学生，或者只能通过学生访问老师。
- 双向关系：关联的两端可以互相访问。例如，老师和学生之间可以互相访问。

单向关联可分为：

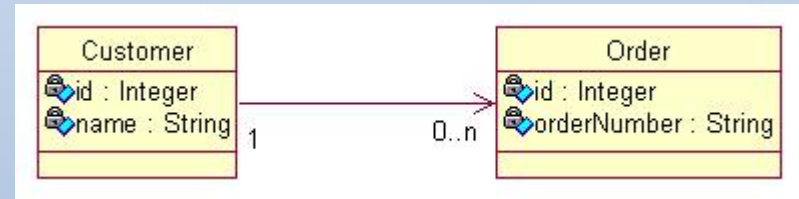
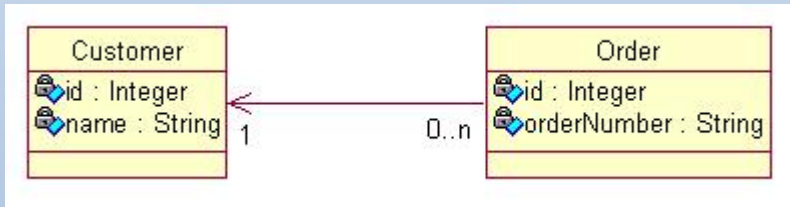
- 单向 1-1
- 单向 1- N
- 单向 N -1
- 单向 N - N

双向关联又可分为：

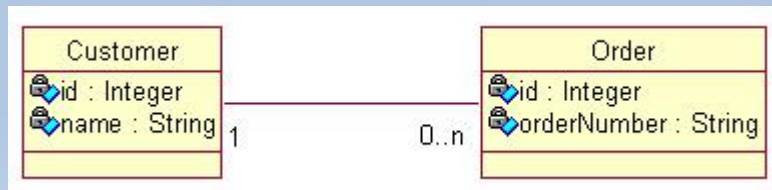
- 双向 1-1
- 双向 1- N
- 双向 N - N

一对多关联关系

- 在领域模型中, 类与类之间最普遍的关系就是关联关系.
- 在 UML 中, 关联是有方向的.
 - 以 Customer 和 Order 为例: 一个用户能发出多个订单, 而一个订单只能属于一个客户. 从 Order 到 Customer 的关联是多对一关联; 而从 Customer 到 Order 是一对多关联
 - 单向关联



- 双向关联

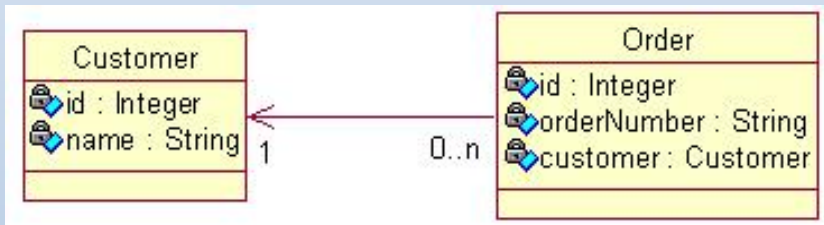


单向 n-1

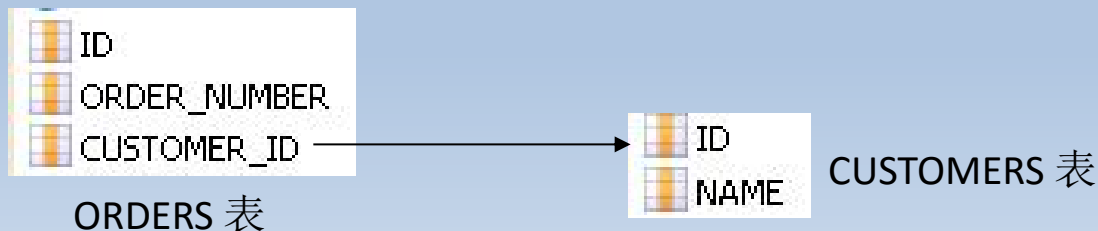
- 单向N-1关系，比如多个人对应一个地址，只需从人实体端可以找到对应的地址实体，无须关系某个地址的全部住户。
- 单向 n-1 关联只需从 n 的一端可以访问 1 的一端。

单向 n-1

- 域模型: 从 Order 到 Customer 的多对一单向关联需要在 Order 类中定义一个 Customer 属性, 而在 Customer 类中无需定义存放 Order 对象的集合属性



- 关系数据模型: ORDERS 表中的 CUSTOMER_ID 参照 CUSTOMER 表的主键



单向 n-1

- 显然无法直接用 property 映射 customer 属性
- Hibernate 使用 <many-to-one> 元素来映射多对一关联关系

```
<many-to-one  
    name="customer"  
    class="Customer"  
    column="CUSTOMER_ID"  
    not-null="true"/>
```

many-to-one

ⓐ name ●
ⓐ access
ⓐ cascade ●
ⓐ class ●
ⓐ column ●
ⓐ embed-xml="true"
ⓐ entity-name
ⓐ fetch="join" ●
ⓐ foreign-key
ⓐ formula
ⓐ index
ⓐ insert="true"
ⓐ lazy="false" ●
ⓐ node
ⓐ not-found="exception"
ⓐ not-null="true"
ⓐ optimistic-lock="true"
ⓐ outer-join="true"
ⓐ property-ref
ⓐ unique-key
ⓐ unique="false"
ⓐ update="true"

- **<many-to-one> 元素来映射组成关系**
 - name: 设定待映射的持久化类的属性的名字
 - column: 设定和持久化类的属性对应的表的外键
 - class : 设定待映射的持久化类的属性的类型

- public class Customer {
- private Integer customerId;
- private String customerName;
- public Integer getCustomerId() {
- return customerId;
- }
- public void setCustomerId(Integer customerId) {
- this.customerId = customerId;
- }
- public String getCustomerName() {
- return customerName;
- }
- public void setCustomerName(String customerName) {
- this.customerName = customerName;
- }
- }

- public class Order {
- private Integer orderId;
- private String orderName;
- private Customer customer;
- public Integer getOrderId() {
- return orderId;
- }
- public void setOrderId(Integer orderId) {
- this.orderId = orderId;
- }
- public String getOrderName() {
- return orderName;
- }
- public void setOrderName(String orderName) {
- this.orderName = orderName;
- }
- public Customer getCustomer() {
- return customer;
- }
- public void setCustomer(Customer customer) {
- this.customer = customer;
- }
- }

Customer.hbm.xml

- `<?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 2015-7-7 14:39:03 by Hibernate Tools 3.4.0.CR1 -->`
- `<hibernate-mapping>`
- `<class name="com.lihui.hibernate.single_n_1.Customer" table="CUSTOMERS">`
- `<id name="customerId" type="java.lang.Integer">`
- `<column name="CUSTOMER_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="customerName" type="java.lang.String">`
- `<column name="CUSTOMER_NAME" />`
- `</property>`
- `</class>`
- `</hibernate-mapping>`

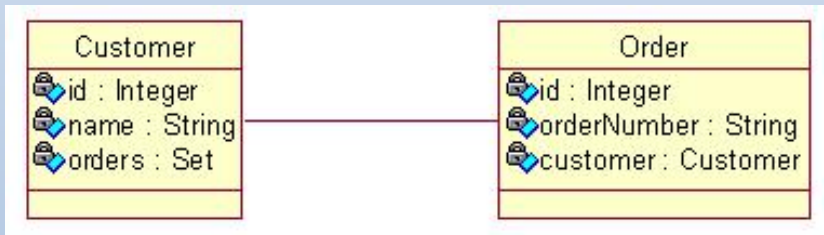
Order.hbm.xml

- `<?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 2015-7-7 14:39:03 by Hibernate Tools 3.4.0.CR1 -->`
- `<hibernate-mapping package="com.lihui.hibernate.single_n_1">`
- `<class name="Order"`
- `table="ORDERS">`
- `<id name="orderId" type="java.lang.Integer">`
- `<column name="ORDER_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="orderName" type="java.lang.String">`
- `<column name="ORDER_NAME" />`
- `</property>`
- `<many-to-one name="customer" class="Customer" column="CUSTOMER_ID"`
- `cascade="all" />`
- `</class>`
- `</hibernate-mapping>`

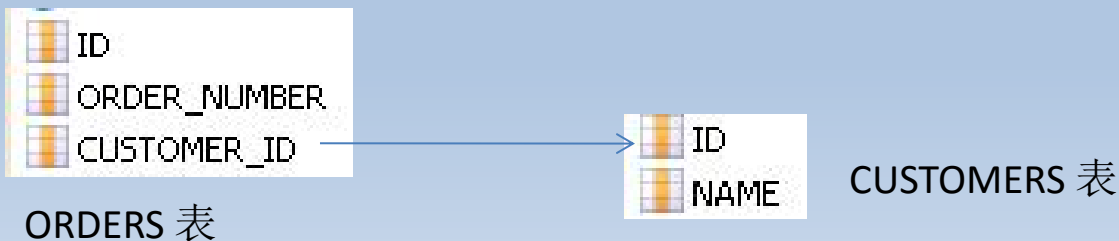
- `public void testSave() {`
- `System.out.println("testSave");`
- `Customer customer = new Customer();`
- `customer.setCustomerName("cc");`
-
- `Order order1 = new Order();`
- `order1.setOrderName("ccc1");`
- `order1.setCustomer(customer);`
- `Order order2 = new Order();`
- `order2.setOrderName("ccc2");`
- `order2.setCustomer(customer);`
-
- `session.save(order1);`
- `session.save(order2);`
- `}`

双向 1-n

- 双向 1-n 与 双向 n-1 是完全相同的两种情形
- 双向 1-n 需要在 1 的一端可以访问 n 的一端, 反之亦然.
- 域模型: 从 Order 到 Customer 的多对一双向关联需要在 Order 类中定义一个 Customer 属性, 而在 Customer 类中需定义存放 Order 对象的集合属性



- 关系数据模型: ORDERS 表中的 CUSTOMER_ID 参照 CUSTOMER 表的主键



双向 1-n

- 当 Session 从数据库中加载 Java 集合时, 创建的是 Hibernate 内置集合类的实例, 因此**在持久化类中定义集合属性时必须把属性声明为 Java 接口类型**
 - Hibernate 的内置集合类具有集合代理功能, **支持延迟检索策略**
 - 事实上, Hibernate 的内置集合类封装了 JDK 中的集合类, 这使得 Hibernate 能够对缓存中的集合对象进行脏检查, 按照集合对象的状态来同步更新数据库。
- 在定义集合属性时, 通常把它初始化为集合实现类的一个实例. 这样可以提高程序的健壮性, 避免应用程序访问取值为 null 的集合的方法抛出 NullPointerException

```
private Set<Order> orders = new HashSet<Order>();  
public Set<Order> getOrders() {  
    return orders;  
}  
public void setOrders(Set<Order> orders) {  
    this.orders = orders;  
}
```

双向 1-n

- Hibernate 使用 <set> 元素来映射 set 类型的属性

```
<set name="orders">  
  <key column="CUSTOMER_ID"></key>  
  <one-to-many class="Order"/>  
</set>
```

set

ⓐ @name ●
ⓐ access
ⓐ batch-size ●
ⓐ cascade
ⓐ catalog
ⓐ check
ⓐ collection-type
ⓐ embed-xml="true"
ⓐ fetch="join" ●
ⓐ inverse="false" ●
ⓐ lazy="true" ●
ⓐ mutable="true"
ⓐ node
ⓐ optimistic-lock="true"
ⓐ order-by
ⓐ outer-join="true"
ⓐ persister
ⓐ schema
ⓐ sort="unsorted" ●
ⓐ subselect ●
ⓐ table
ⓐ where

- <set> 元素来映射持久化类的 set 类型的属性
 - name: 设定待映射的持久化类的属性的

key

- ③ column ●
- ③ foreign-key
- ③ not-null="true"
- ③ on-delete="noaction"
- ③ property-ref
- ③ unique="true"
- ③ update="true"

- **<key>** 元素设定与所关联的持久化类对应的表的外键
 - column: 指定关联表的外键名

one-to-many

④ class ●
④ embed-xml
④ entity-name
④ node
④ not-found

- **<one-to-many>** 元素设定集合属性中所关联的持久化类
 - **class:** 指定关联的持久化类的类名


```
<class name="Order" table="ORDERS">  
  <id name="orderId" type="java.lang.Integer">  
    <column name="ORDER_ID" />  
    <generator class="native" />  
  </id>
```

```
<many-to-one name="customer"  
  class="Customer"  
  column="CUSTOMER_ID"  
  lazy="proxy"></many-to-one>
```

```
<set name="orders" table="ORDERS">  
  <key column="CUSTOMER_ID"></key>  
  <one-to-many class="Order"/>  
</set>
```

<set> 元素的 inverse 属性

- 在hibernate中通过对 inverse 属性的来决定是由双向关联的哪一方来维护表和表之间的关系. inverse = false 的为主动方, inverse = true 的为被动方, 由主动方负责维护关联关系
- 在没有设置 inverse=true 的情况下, 父子两边都维护父子关系
- 在 1-n 关系中, 将 n 方设为主控方将有助于性能改善(如果要国家元首记住全国人民的姓名, 不是太可能, 但要让全国人民知道国家元首, 就容易的多)
- 在 1-N 关系中, 若将 1 方设为主控方
 - 会额外多出 update 语句。
 - 插入数据时无法同时插入外键列, 因而无法为外键列添加非空约束

cascade 属性

- 在对象 – 关系映射文件中, 用于映射持久化类之间关联关系的元素, <set>, <many-to-one> 和 <one-to-one> 都有一个 cascade 属性, 它用于指定如何操纵与当前对象关联的其他对象.

cascade属性值	描述
none	当Session操纵当前对象时, 忽略其他关联的对象。它是cascade属性的默认值
save-update ●	当通过Session的save()、update()及saveOrUpdate()方法来保存或更新当前对象时, 级联保存所有关联的新建的临时对象, 并且级联更新所有关联的游离对象
persist	当通过Session的persist()方法来保存当前对象时, 会级联保存所有关联的新建的临时对象
merge	当通过Session的merge()方法来保存当前对象时, 会级联融合所有关联的游离对象
delete ●	当通过Session的delete()方法删除当前对象时, 会级联删除所有关联的对象
lock	当通过Session的lock()方法把当前游离对象加入到Session缓存中时, 会把所有关联的游离对象也加入到Session缓存中。
replicate	当通过Session的replicate()方法复制当前对象时, 会级联复制所有关联的对象
evict	当通过Session的evict()方法从Session缓存中清除当前对象时, 会级联清除所有关联的对象
refresh	当通过Session的refresh()方法刷新当前对象时, 会级联刷新所有关联的对象。所谓刷新是指读取数据库中相应数据, 然后根据数据库中的最新数据去同步更新Session缓存中的相应对象
all	包含save-update、persist、merge、delete、lock、replicate、evict及refresh的行为
delete-orphan ●	删除所有和当前对象解除关联关系的对象
all-delete-orphan ●	包含all和delete-orphan的行为

- public class Order {
- private Integer orderId;
- private String orderName;
- private Customer customer;
- ...
- }

- public class Customer {
- private Integer customerId;
- private String customerName;
- private Set<Order> orders = new HashSet<Order>();
- public Set<Order> getOrders() {
- return orders;
- }
- ...

Order.hbm.xml

- `<?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 2015-7-7 15:40:17 by Hibernate Tools 3.4.0.CR1 -->`
- `<hibernate-mapping package="com.lihui.hibernate.double_n_1">`
- `<class name="Order" table="ORDERS">`
- `<id name="orderId" type="java.lang.Integer">`
- `<column name="ORDER_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="orderName" type="java.lang.String">`
- `<column name="ORDER_NAME" />`
- `</property>`
- `<many-to-one name="customer" class="Customer" cascade="all"`
- `column="CUSTOMER_ID"></many-to-one>`
- `</class>`
- `</hibernate-mapping>`

Customer.hbm.xml

- `<?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 2015-7-7 15:40:17 by Hibernate Tools 3.4.0.CR1 -->`
- `<hibernate-mapping package="com.lihui.hibernate.double_n_1">`
- `<class name="Customer" table="CUSTOMERS">`
- `<id name="customerId" type="java.lang.Integer">`
- `<column name="CUSTOMER_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="customerName" type="java.lang.String">`
- `<column name="CUSTOMER_NAME" />`
- `</property>`
- `<set name="orders" inverse="true" cascade="all">`
- `<key column="CUSTOMER_ID"></key>`
- `<one-to-many class="Order"/>`
- `</set>`
- `</class>`
- `</hibernate-mapping>`

- Customer customer = new Customer();
- customer.setCustomerName("ckk");
- Order order1 = new Order();
- order1.setOrderName("Ckkk");
- order1.setCustomer(customer);
- Order order2 = new Order();
- order2.setOrderName("Dkkk");
- order2.setCustomer(customer);
- session.save(order1);
- session.save(order2);

- Customer customer = new Customer();
- customer.setCustomerName("yyy");
-
- Order order1 = new Order();
- order1.setOrderName("yykkk");
-
- Order order2 = new Order();
- order2.setOrderName("dddkkk");
-
- customer.getOrders().add(order1);
- customer.getOrders().add(order2);
-
- session.save(customer);

在数据库中对集合排序

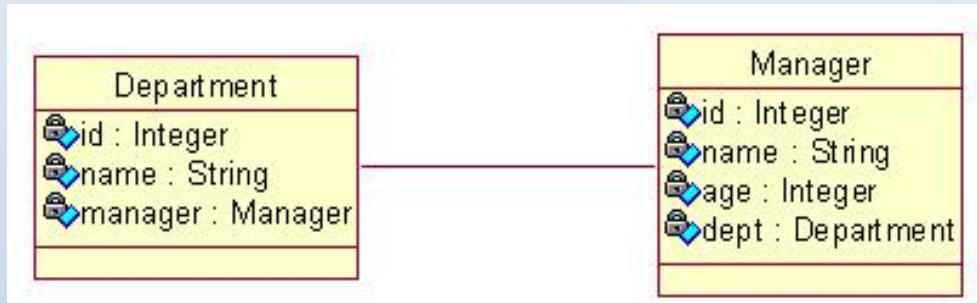
- <set> 元素有一个 order-by 属性, 如果设置了该属性, 当 Hibernate 通过 select 语句到数据库中检索集合对象时, 利用 order by 子句进行排序
- order-by 属性中还可以加入 SQL 函数

```
<set name="orders" inverse="true" cascade="save-update" order-by="ORDER_DATE">  
  <key column="CUSTOMER_ID"></key>  
  <one-to-many class="Order"/>  
</set>
```

映射一对一关联关系

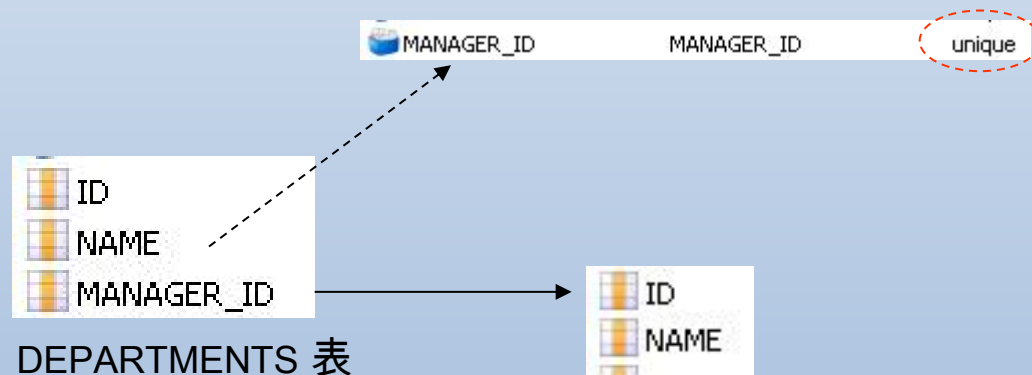
1 - 1

- 域模型

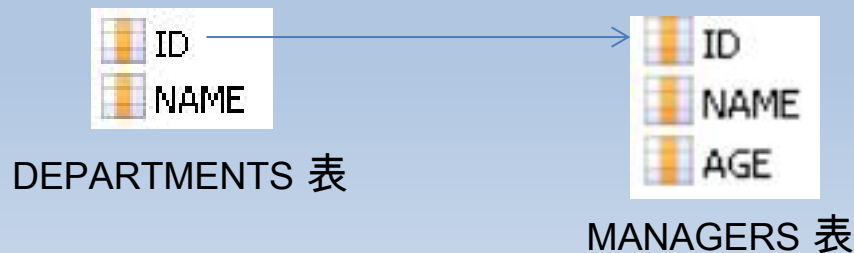


- 关系数据模型:

- 按照外键映射:



- 按照主键映射:



基于外键映射的 1-1

- 对于基于外键的1-1关联，其外键可以存放在任意一边，**在需要存放外键一端，增加many-to-one元素**。为many-to-one元素增加unique="true" 属性来表示为1-1关联

```
<many-to-one name="manager" class="Manager" column="MANAGER_ID"
    cascade="all" unique="true" />
```

- 另一端需要使用one-to-one元素，该元素使用 **property-ref** 属性指定使用被关联实体主键以外的字段作为关联字段

```
<one-to-one name="dept" class="Department" property-ref="manager" />
```

- 不使用 property-ref 属性的 sql

```
from MANAGERS manager0_
left outer join DEPARTMENTS department1_
on manager0_.ID=department1_.ID
where manager0_.ID=?
```

- 使用 property-ref 属性的 sql

```
from MANAGERS manager0_
left outer join DEPARTMENTS department1_
on manager0_.ID=department1_.MANAGER_ID
where manager0_.ID=?
```

Department.java

- public class Department {
- private Integer deptId;
- private String deptName;
- private Manager manager;
- ...
- }

Manager.java

- public class Manager {
- private Integer mgrId;
- private String mgrName;
- private Department department;
- ...
- }

Department.hbm.xml

- `<?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 2015-7-7 19:33:37 by Hibernate Tools 3.4.0.CR1 -->`
- `<hibernate-mapping package="com.lihui.hibernate.double_1_1.foreign">`
- `<class name="Department" table="DEPARTMENTS">`
- `<id name="deptId" type="java.lang.Integer">`
- `<column name="DEPT_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="deptName" type="java.lang.String">`
- `<column name="DEPT_NAME" />`
- `</property>`
- `<many-to-one name="manager" class="Manager" cascade="all" column="MGR_ID"`
- `unique="true"></many-to-one>`
- `</class>`
- `</hibernate-mapping>`

Manager.hbm.xml

- `<?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 2015-7-7 19:33:37 by Hibernate Tools 3.4.0.CR1 -->`
- `<hibernate-mapping package="com.lihui.hibernate.double_1_1.foreign">`
- `<class name="Manager" table="MANAGERS">`
- `<id name="mgrId" type="java.lang.Integer">`
- `<column name="MGR_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="mgrName" type="java.lang.String">`
- `<column name="MGR_NAME" />`
- `</property>`
- `<!-- 映射1-1的关联关系，在对应的数据表中已经有外键，当前持久花类使用ont-to-one进行映射 -->`
- `<one-to-one name="department" class="Department" property-ref="manager"></one-to-one>`
- `</class>`
- `</hibernate-mapping>`

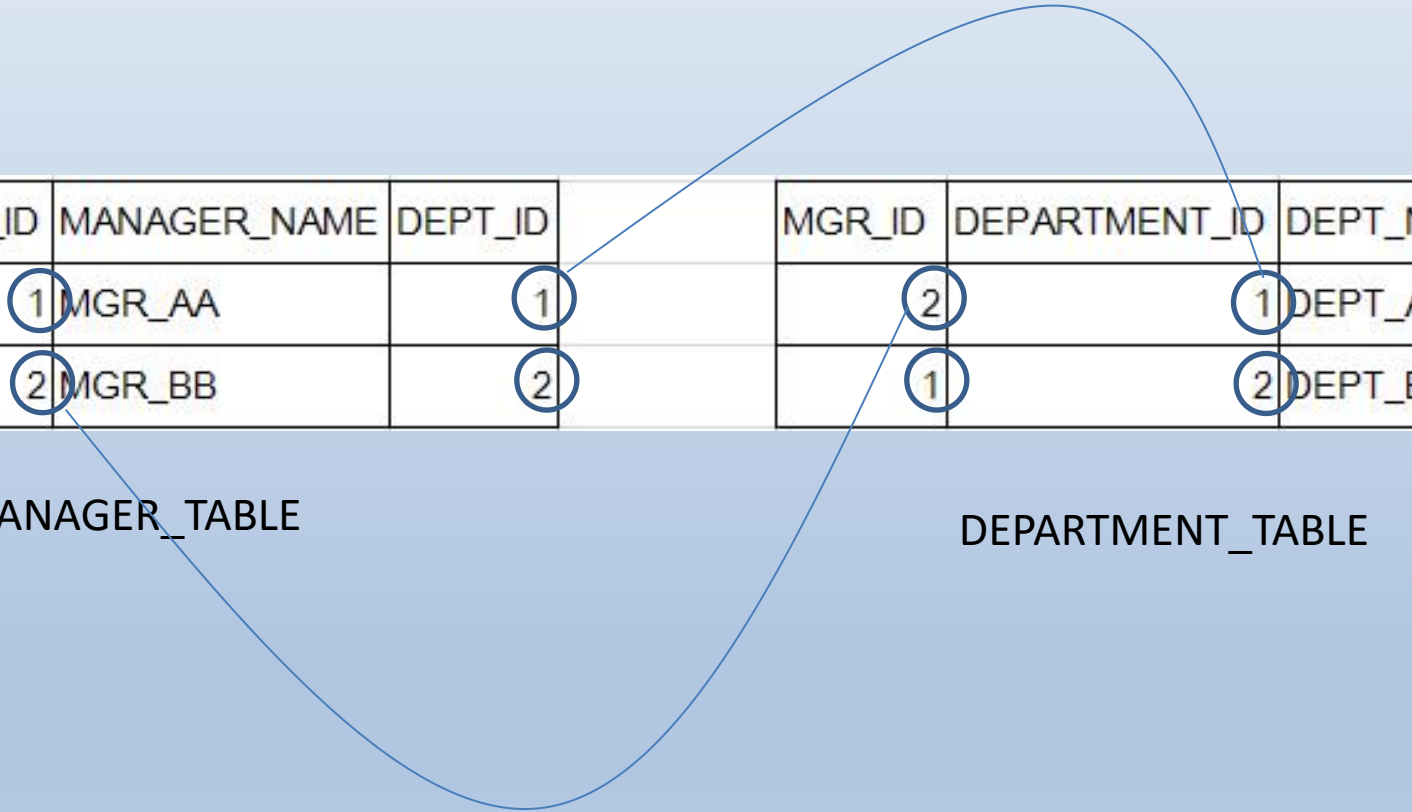
- @Test
- public void testGet(){
- Department dep = (Department) session.get(Department.class, 1);
- System.out.println(dep.getDeptName());
- Manager mgr = dep.getManager();
- System.out.println(mgr.getMgrName());
- }
- @Test
- public void testSave() {
- try{
- Department dep = new Department();
- dep.setDeptName("aaa");
- Manager mgr = new Manager();
- mgr.setMgrName("bbb");
- dep.setManager(mgr);
- mgr.setDepartment(dep);
- session.save(mgr);
- session.save(dep);
- }
- catch(Exception e){
- e.printStackTrace();
- }
- System.out.println("aaa");
- }

两边都使用外键映射的 1-1

MANAGER_ID	MANAGER_NAME	DEPT_ID		MGR_ID	DEPARTMENT_ID	DEPT_NAME
1	MGR_AA	1		2	1	DEPT_AA
2	MGR_BB	2		1	2	DEPT_BB

MANAGER_TABLE

DEPARTMENT_TABLE



基于主键映射的 1-1

- 基于主键的映射策略:指一端的主键生成器使用 foreign 策略,表明根据“对方”的主键来生成自己的主键,自己并不能独立生成主键. <param> 子元素指定使用当前持久化类的哪个属性作为 “对方”

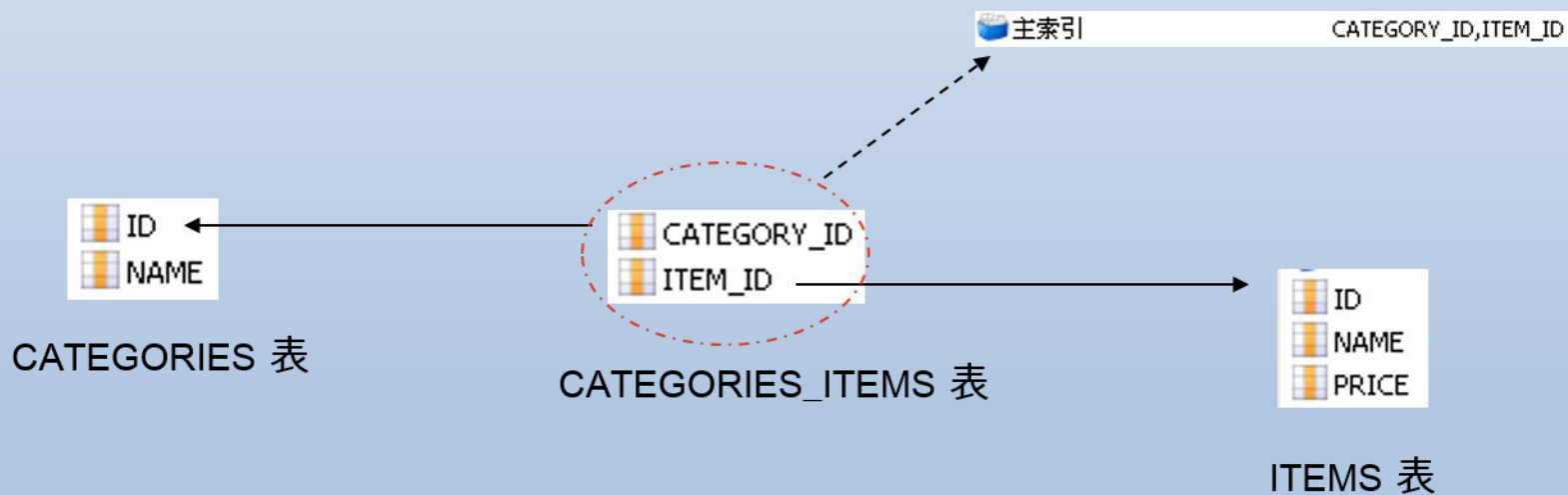
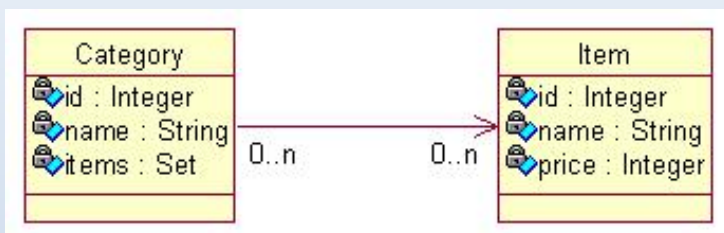
```
<id name="id" column="ID" type="integer">  
  <generator class="foreign">  
    <param name="property">manager</param>  
  </generator>  
</id>
```

- 采用foreign主键生成器策略的一端增加 one-to-one 元素映射关联属性,其 one-to-one属性还应增加 constrained="true" 属性;另一端增加one-to-one 元素映射关联属性。
- constrained**(约束):指定为当前持久化类对应的数据库表的主键添加一个外键约束,引用被关联的对象(“对方”)所对应的数据库表主键

```
<one-to-one  
  name="manager"  
  class="Manager"  
  constrained="true"/>
```

单向N-N关联

- N-N关联映射增加一张表才完成基本映射。
- 与1-N映射相似，必须为set集合元素添加key子元素，指定CATEGORIES_ITEMS表中参照CATEGORIES表的外键为CATEGORIY_ID。
- 与1-N不同的是，建立N-N关联时，集合中的元素使用many-to-many。关于配置文件的属性的介绍，将在代码实现部分介绍。



- public class Category {
- private Integer categoryId;
- private String categoryName;
- private Set<Item> items = new HashSet<Item>();
- //省去get和set方法
- }
- public class Item {
- private Integer itemId;
- private String itemName;
- //省去get和set方法
- }

Category.hbm.xml

-
- `<hibernate-mapping package="com.lihui.hibernate.single_n_n">`
- `<class name="Category" table="CATEGORIES">`
- `<id name="categoryId" type="java.lang.Integer">`
- `<column name="CATEGORY_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="categoryName" type="java.lang.String">`
- `<column name="CATEGORY_NAME" />`
- `</property>`
- `<set name="items" table="CATEGORIES_ITEMS">`
- `<key>`
- `<column name="C_ID"></column>`
- `</key>`
- `<many-to-many class="Item" column="I_ID"></many-to-many>`
- `</set>`
- `</class>`
- `</hibernate-mapping>`

table:指定中间表
many-to-many:指定多对多的关联关系
column:执行set集合中的持久化类在中间表的外键列的名称

Item.hbm.xml

-
- `<hibernate-mapping package="com.lihui.hibernate.single_n_n">`
- `<class name="Item" table="ITEMS">`
- `<id name="itemId" type="java.lang.Integer">`
- `<column name="ITEM_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="itemName" type="java.lang.String">`
- `<column name="ITEM_NAME" />`
- `</property>`
- `</class>`
- `</hibernate-mapping>`

- @Test
- public void testSave() {
- Category c1 = new Category();
- c1.setCatregoryName("C-AA");
- Category c2 = new Category();
- c2.setCatregoryName("C-BB");
- Item i1 = new Item();
- i1.setItemName("I-AA");
- Item i2 = new Item();
- i2.setItemName("I-BB");
- c1.getItems().add(i1);
- c1.getItems().add(i2);
- c2.getItems().add(i1);
- c2.getItems().add(i2);
- session.save(c1);
- session.save(c2);
- session.save(i1);
- session.save(i2);
- }

双向N-N关联

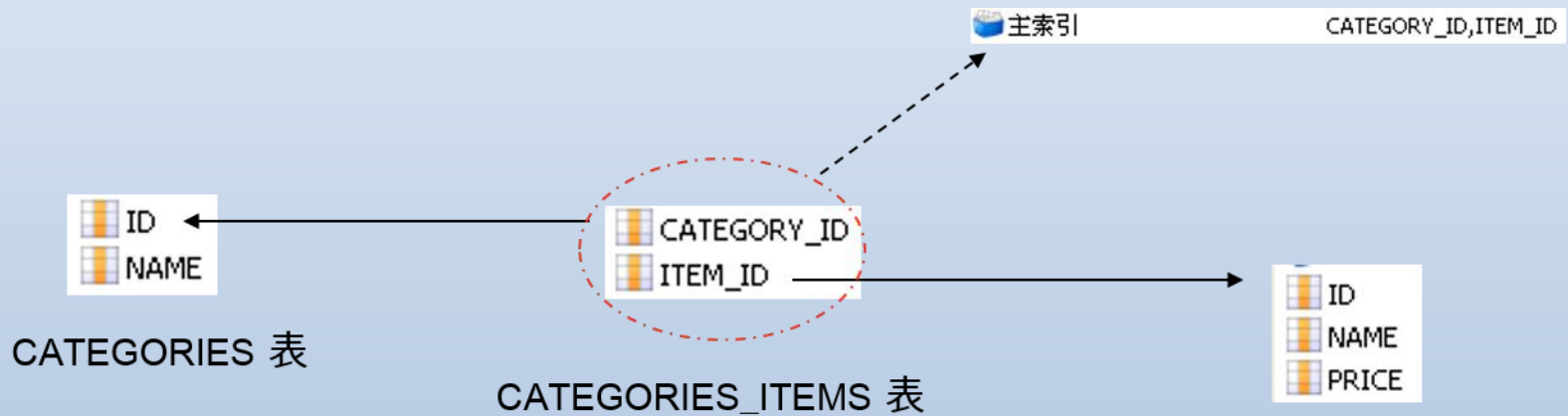
- 双向N-N关联需要两端都使用set集合属性，两端都增加对集合属性的访问。
- 在双向N-N关联的两边都需指定连接表的表名及外键列的列名. 两个集合元素 set 的 table 元素的值必须指定，而且必须相同。set元素的两个子元素：key 和 many-to-many 都必须指定 column 属性，其中，key 和 many-to-many 分别指定本持久化类和关联类在连接表中的外键列名，因此两边的 key 与 many-to-many 的column属性交叉相同。也就是说，一边的set元素的key的 cloumn值为a,many-to-many 的 column 为b；则另一边的 set 元素的 key 的 column 值 b,many-to-many的 column 值为 a

- 对于双向 n-n 关联, 必须把其中一端的 `inverse` 设置为 `true`, 否则两端都维护关联关系可能会造成主键冲突

```
<set name="items"  
  table="ITEM_CATEGORY">  
  
  <key column="CATEGORY_ID"></key>  
  <many-to-many class="Item"  
    column="ITEM_ID"></many-to-many>  
</set>
```

```
<set name="categories"  
  table="ITEM_CATEGORY"  
  inverse="true">  
  
  <key column="ITEM_ID"></key>  
  <many-to-many class="Category"  
    column="CATEGORY_ID"></many-to-many>  
</set>
```

关系数据模型



-
- public class Category {
- private Integer categoryId;
- private String categoryName;
- private Set<Item> items = new HashSet<Item>();
- //省去get和set方法
- }
- public class Item {
- private Integer itemId;
- private String itemName;
- //省去get和set方法
- }

Category.hbm.xml

-
- `<hibernate-mapping package="com.lihui.hibernate.single_n_n">`
- `<class name="Category" table="CATEGORIES">`
- `<id name="categoryId" type="java.lang.Integer">`
- `<column name="CATEGORY_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="categoryName" type="java.lang.String">`
- `<column name="CATEGORY_NAME" />`
- `</property>`
- `<set name="items" table="CATEGORIES_ITEMS">`
- `<key>`
- `<column name="C_ID"></column>`
- `</key>`
- `<many-to-many class="Item" column="I_ID"></many-to-many>`
- `</set>`
- `</class>`
- `</hibernate-mapping>`

table:指定中间表
many-to-many:指定多对多的关联关系
column:执行set集合中的持久化类在中间表的外键列的名称

Item.hbm.xml

- `<hibernate-mapping package="com.lihui.hibernate.single_n_n">`
- `<class name="Item" table="ITEMS">`
- `<id name="itemId" type="java.lang.Integer">`
- `<column name="ITEM_ID" />`
- `<generator class="native" />`
- `</id>`
- `<property name="itemName" type="java.lang.String">`
- `<column name="ITEM_NAME" />`
- `</property>`
- `<set name="categories" table="CATEGORIES_ITEMS" inverse="true">`
- `<key>`
- `<column name="I_ID"></column>`
- `</key>`
- `<many-to-many class="Category" column="C_ID"></many-to-many>`
- `</set>`
- `</class>`
- `</hibernate-mapping>`
- 注意要在其中一端加入`inverse="true"`，否则会造成主键冲突