



Hibernate

讲师：佟刚

新浪微博：尚硅谷-佟刚

Hello Hibernate

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).

ORM

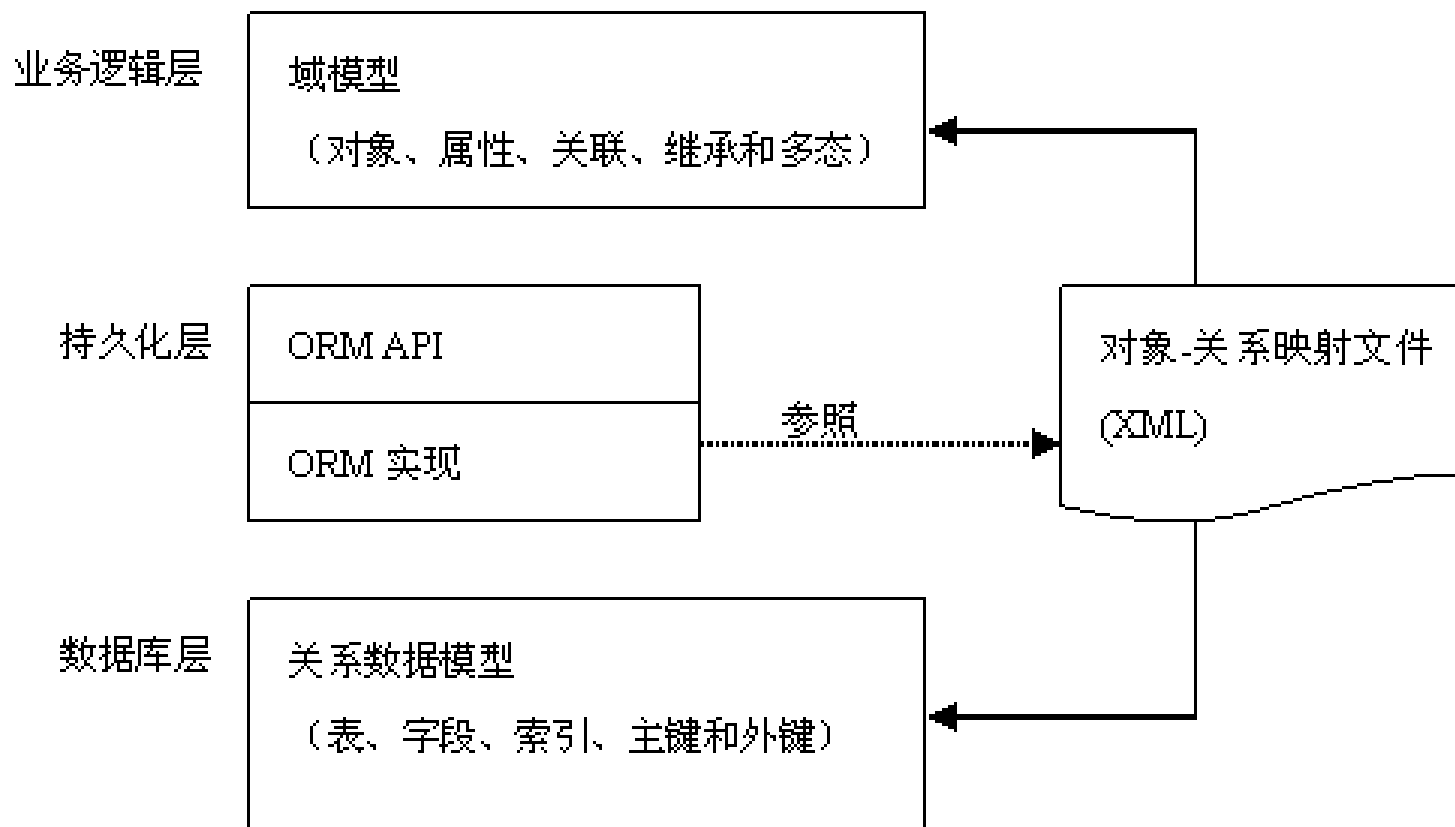
- ORM(Object/Relation **Mapping**): **对象/关系映射**

- 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



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);  
}
```

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

安装 hibernate 插件

- 安装方法说明 (hibernatetools-4.1.1.Final) :
 - **Help --> Install New Software...**
 - Click Add...
 - In dialog Add Site dialog, click **Archive...**
 - Navigate to **hibernatetools-Update-4.1.1.Final_2013-12-08_01-06-33-B605.zip** and click **Open**
 - Clicking **OK** in the Add Site dialog will bring you back to the dialog 'Install'
 - Select the **Jboss Tools hibernatetools Nightly Build Update Site** that has appeared
 - Click **Next** and then **Finish**
 - **Approve the license**
 - Restart eclipse when that is asked

准备 Hibernate 环境

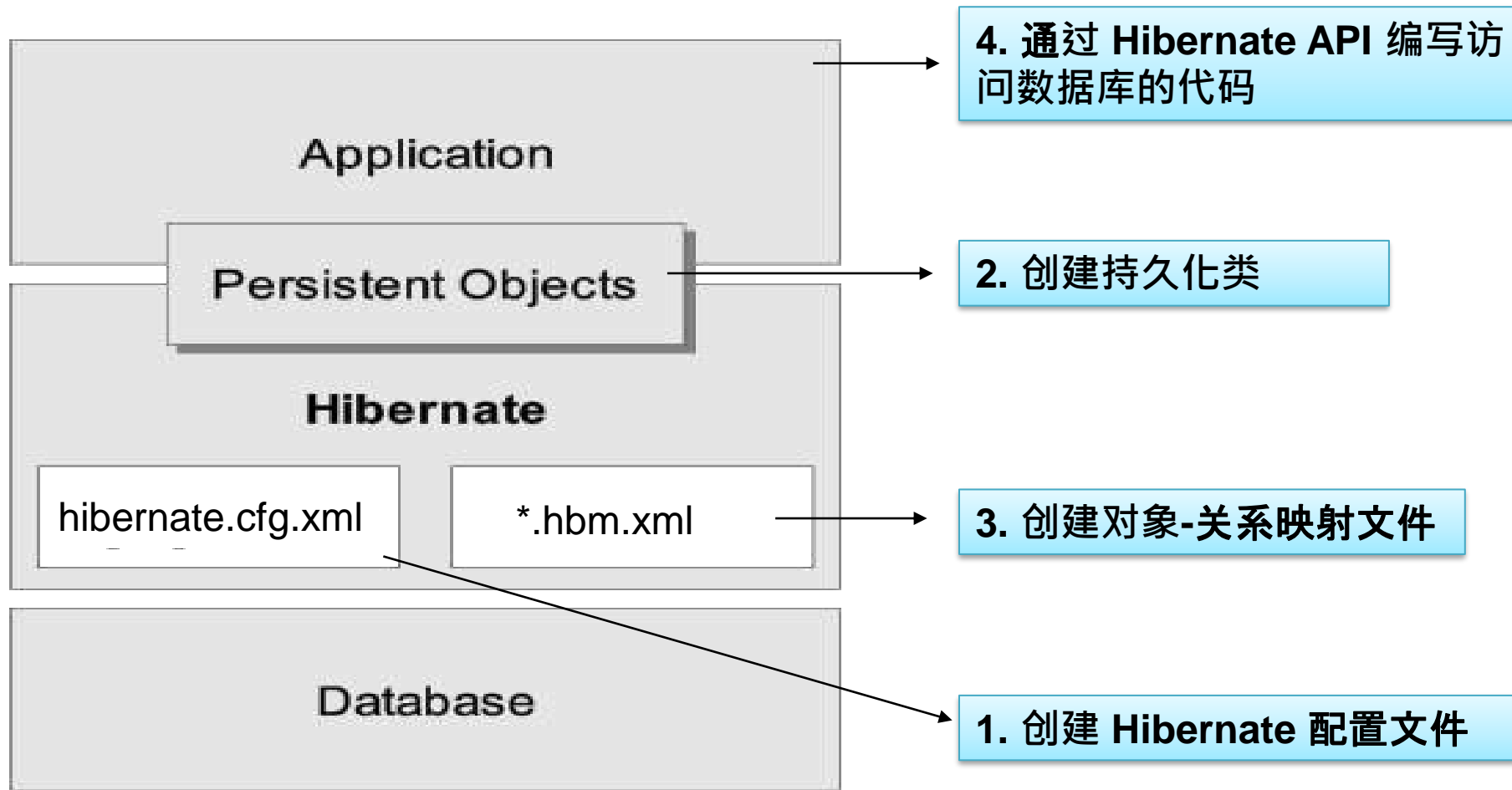
- 导入 Hibernate 必须的 jar 包:

-  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 :

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

Hibernate开发步骤

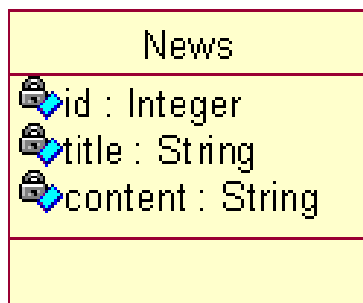


1. 创建持久化 Java 类

- **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

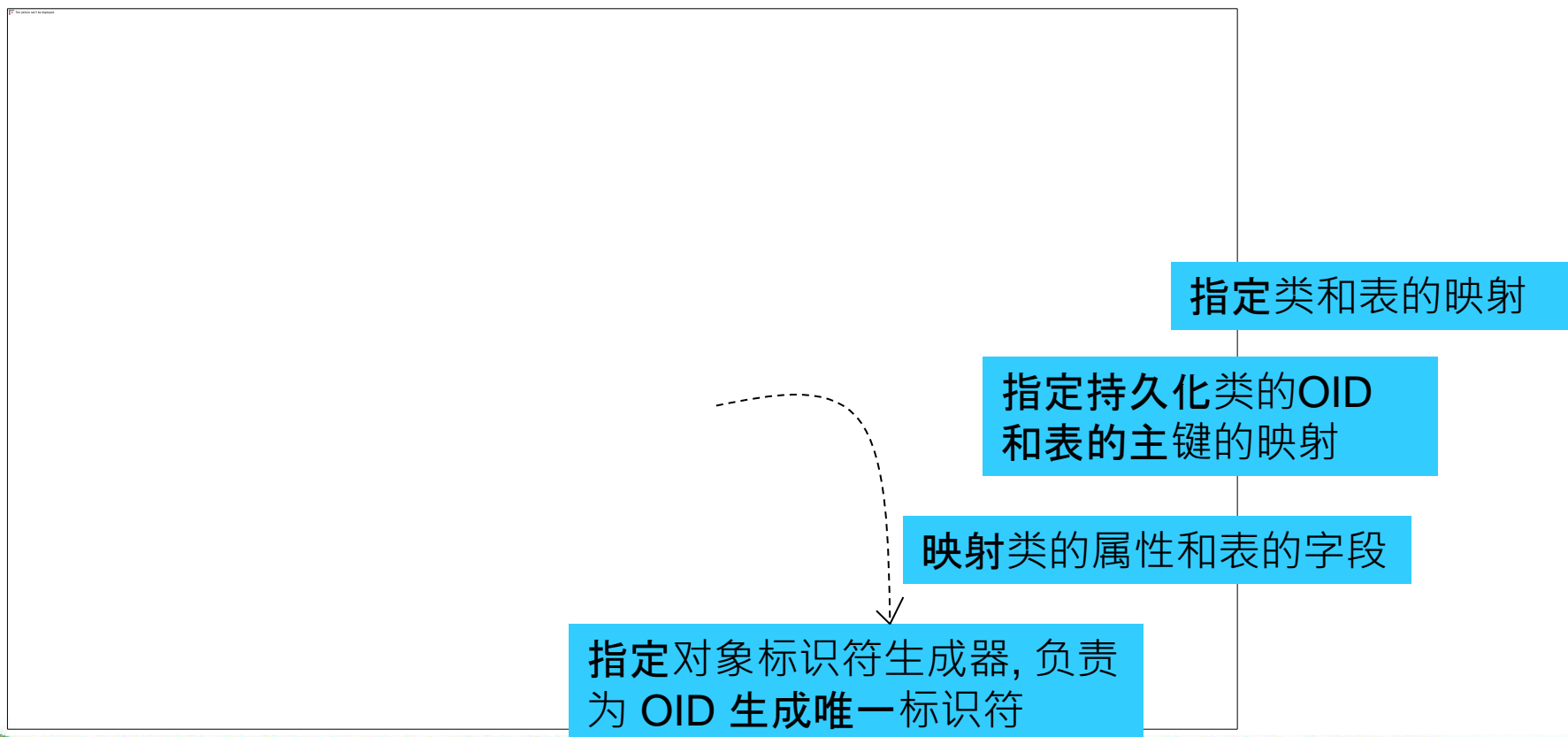
1. 创建持久化 Java 类

- **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



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



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

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

ServiceRegistry serviceRegistry =
    new ServiceRegistryBuilder().applySettings(configuration.getProperties())
                                .buildServiceRegistry();

SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();

Customer customer = new Customer();
customer.setBirth(new Date(new java.util.Date().getTime()));
customer.setCustomerName("ATGUIGU");
session.save(customer);

transaction.commit();
session.close();
sessionFactory.close();
```

- 下边是控制台输出的 SQL 语句

```
insert into CUSTOMERS (CUSTOMER_NAME, BIRTH) values (?, ?)
```


Helloworld

- 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

讲师：佟刚

新浪微博：尚硅谷-佟刚

Session 概述

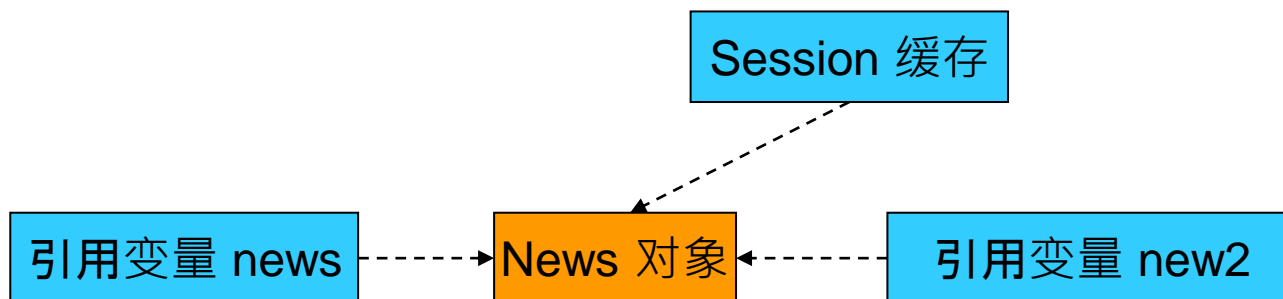
- Session 接口是 Hibernate 向应用程序提供的操纵数据库的最主要的接口, 它**提供了基本的保存, 更新, 删除和加载 Java 对象的方法**.
- **Session 具有一个缓存, 位于缓存中的对象称为持久化对象, 它和数据库中的相关记录对应**. Session 能够在某些时间点, 按照缓存中对象的变化来执行相关的 SQL 语句, 来同步更新数据库, 这一过程被称为刷新缓存(flush)
- **站在持久化的角度, Hibernate 把对象分为 4 种状态**: 持久化状态, 临时状态, 游离状态, 删除状态. Session 的特定方法能使对象从一个状态转换到另一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

Session 缓存

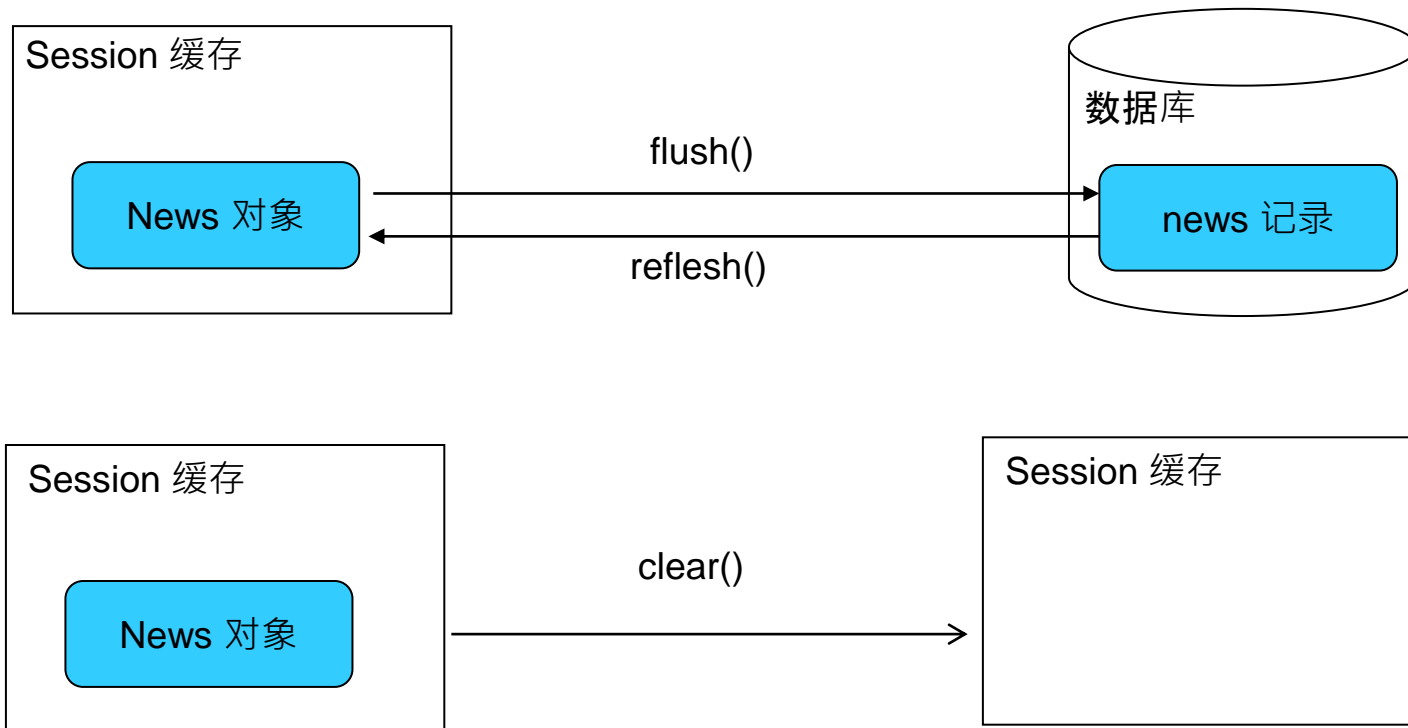
- 在 Session 接口的实现中包含一系列的 Java 集合, 这些 Java 集合构成了 Session 缓存. 只要 Session 实例没有结束生命周期, 且没有清理缓存, 则存放在它缓存中的对象也不会结束生命周期
- Session 缓存可减少 Hibernate 应用程序访问数据库的频率。

```
41 News news = (News) session.get(News.class, 1);  
42 System.out.println(news);  
43  
44 News news2 = (News) session.get(News.class, 1);  
45 System.out.println(news2);  
46  
47 System.out.println(news == news2);
```

会向数据库发送几条 SQL ?



操作 Session 缓存



flush 缓存

- flush : Session 按照缓存中对象的属性变化来同步更新数据库
- 默认情况下 Session 在以下时间点刷新缓存：
 - 显式调用 Session 的 flush() 方法
 - 当应用程序调用 Transaction 的 commit () 方法的时, 该方法先 flush , 然后在向数据库提交事务
 - 当应用程序执行一些查询(HQL, Criteria)操作时, 如果缓存中持久化对象的属性已经发生了变化, 会先 flush 缓存, 以保证查询结果能够反映持久化对象的最新状态
- flush 缓存的例外情况: 如果对象使用 native 生成器生成 OID, 那么当调用 Session 的 save() 方法保存对象时, 会立即执行向数据库插入该实体的 insert 语句.
- commit() 和 flush() 方法的区别 : flush 执行一系列 sql 语句, 但不提交事务 ; commit 方法先调用flush() 方法, 然后提交事务. 意味着提交事务意味着对数据库操作永久保存下来。

Hibernate 主键生成策略

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。

设定刷新缓存的时间点

- 若希望改变 flush 的默认时间点, 可以通过 Session 的 setFlushMode() 方法显式设定 flush 的时间点

清理缓存的模式	各种查询方法	Transaction 的 commit() 方法	Session 的 flush() 方法
FlushMode.AUTO(默认模式)	清理	清理	清理
FlushMode.COMMIT	不清理	清理	清理
FlushMode.NEVER	不清理	不清理	清理

数据库的隔离级别

- 对于同时运行的多个事务, 当这些事务访问数据库中相同的数据时, 如果没有采取必要的隔离机制, 就会导致各种并发问题:
 - **脏读**: 对于两个事物 T1, T2, T1 读取了已经被 T2 更新但还没有被提交的字段. 之后, 若 T2 回滚, T1 读取的内容就是临时且无效的.
 - **不可重复读**: 对于两个事物 T1, T2, T1 读取了一个字段, 然后 T2 更新了该字段. 之后, T1 再次读取同一个字段, 值就不同了.
 - **幻读**: 对于两个事物 T1, T2, T1 从一个表中读取了一个字段, 然后 T2 在该表中插入了一些新的行. 之后, 如果 T1 再次读取同一个表, 就会多出几行.
- 数据库事务的隔离性: 数据库系统必须具有隔离并发运行各个事务的能力, 使它们不会相互影响, 避免各种并发问题.
- 一个事务与其他事务隔离的程度称为隔离级别. 数据库规定了多种事务隔离级别, 不同隔离级别对应不同的干扰程度, 隔离级别越高, 数据一致性就越好, 但并发性越弱

数据库的隔离级别

- 数据库提供的 4 种事务隔离级别:

隔离级别	描述
READ UNCOMMITTED (读未提交数据)	允许事务读取未被其他事物提交的变更. 脏读, 不可重复读和幻读的问题都会出现
READ COMMITED (读已提交数据)	只允许事务读取已经被其它事务提交的变更. 可以避免脏读, 但不可重复读和幻读问题仍然可能出现
REPEATABLE READ (可重复读)	确保事务可以多次从一个字段中读取相同的值. 在这个事务持续期间, 禁止其他事物对这个字段进行更新. 可以避免脏读和不可重复读, 但幻读的问题仍然存在.
SERIALIZABLE(串行化)	确保事务可以从一个表中读取相同的行. 在这个事务持续期间, 禁止其他事务对该表执行插入, 更新和删除操作. 所有并发问题都可以避免, 但性能十分低下.

- Oracle 支持的 2 种事务隔离级别: **READ COMMITED**, **SERIALIZABLE**. Oracle 默认的事务隔离级别为: **READ COMMITED**
- Mysql 支持 4 中事务隔离级别. Mysql 默认的事务隔离级别为: **REPEATABLE READ**

在 MySQL 中设置隔离级别

- 每启动一个 mysql 程序, 就会获得一个单独的数据库连接. 每个数据库连接都有一个全局变量 @@tx_isolation, 表示当前的事务隔离级别. MySQL 默认的隔离级别为 Repeatable Read
- 查看当前的隔离级别: `SELECT @@tx_isolation;`
- 设置当前 mySQL 连接的隔离级别:
 - set transaction isolation level **read committed**;
- 设置数据库系统的全局的隔离级别:
 - set **global** transaction isolation level **read committed**;

在 Hibernate 中设置隔离级别

- JDBC 数据库连接使用数据库系统默认的隔离级别. 在 Hibernate 的配置文件中可以显式的设置隔离级别. 每一个隔离级别都对应一个整数:
 - 1. READ UNCOMMITTED
 - 2. READ COMMITTED
 - 4. REPEATABLE READ
 - 8. SERIALIZABLE
- Hibernate 通过为 Hibernate 映射文件指定 `hibernate.connection.isolation` 属性来设置事务的隔离级别

持久化对象的状态

- 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.

持久化对象的状态

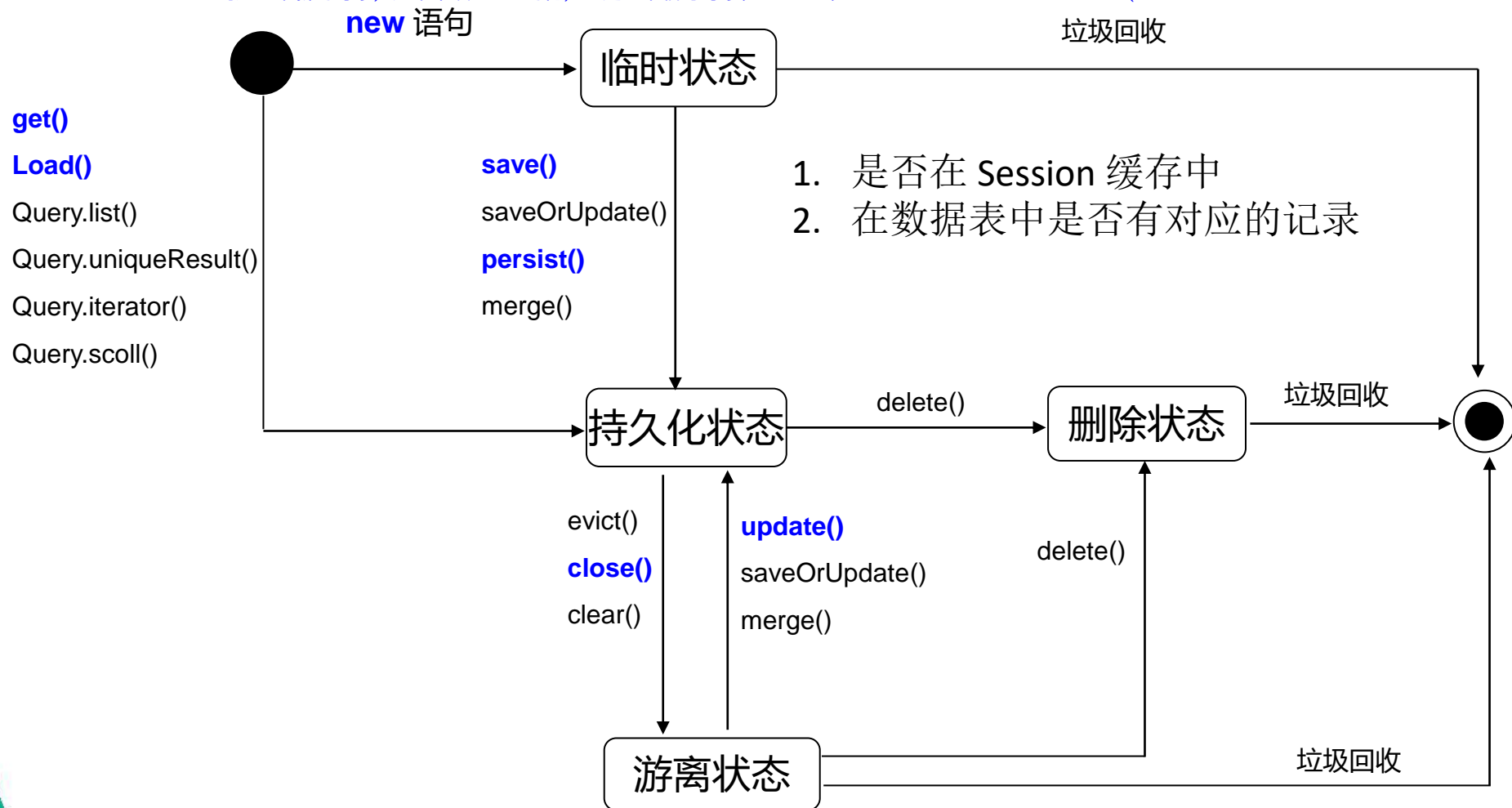
- 临时对象 (Transient) :
 - 在使用代理主键的情况下, **OID 通常为 null**
 - **不处于 Session 的缓存中**
 - **在数据库中没有对应的记录**
- 持久化对象(也叫”托管”) (Persist) :
 - **OID 不为 null**
 - **位于 Session 缓存中**
 - 若在数据库中已经有和其对应的记录, **持久化对象和数据库中的相关记录对应**
 - **Session 在 flush 缓存时, 会根据持久化对象的属性变化, 来同步更新数据库**
 - **在同一个 Session 实例的缓存中, 数据库表中的每条记录只对应唯一的持久化对象**

持久化对象的状态

- 删除对象(Removed)
 - 在数据库中没有和其 OID 对应的记录
 - 不再处于 Session 缓存中
 - 一般情况下, 应用程序不该再使用被删除的对象
- 游离对象(也叫“脱管”) (Detached) :
 - **OID 不为 null**
 - **不再处于 Session 缓存中**
 - 一般情况需下, 游离对象是由持久化对象转变过来的, 因此在数据库中可能还存在与它对应的记录

对象的状态转换图

创建一个新的对象，如果不设置OID的化，它是一个临时对象(Transient)



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.

Session 的 get() 和 load() 方法

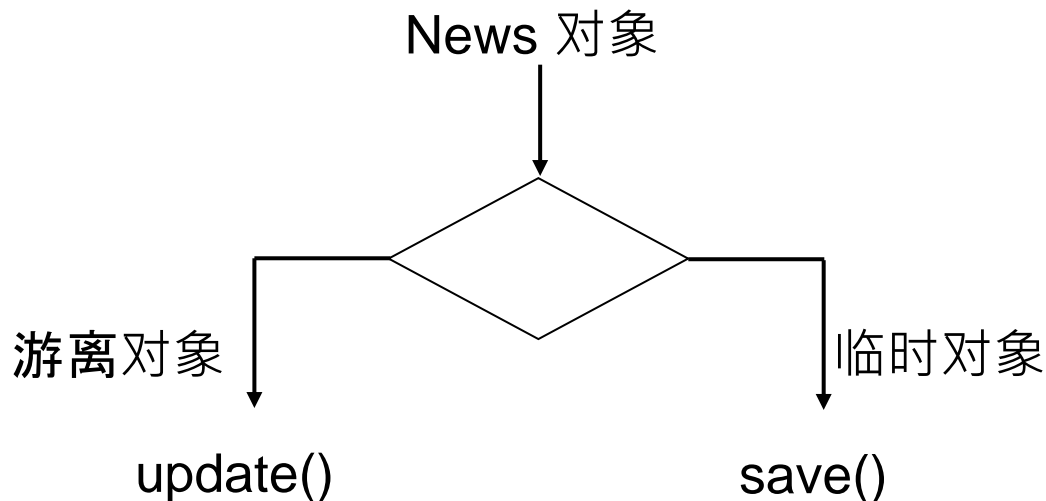
- 都可以根据跟定的 OID 从数据库中加载一个持久化对象
- 区别:
 - 当数据库中不存在与 OID 对应的记录时, load() 方法抛出 ObjectNotFoundException 异常, 而 get() 方法返回 null
 - 两者采用不同的延迟检索策略: load 方法支持延迟加载策略。而 get 不支持。

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.

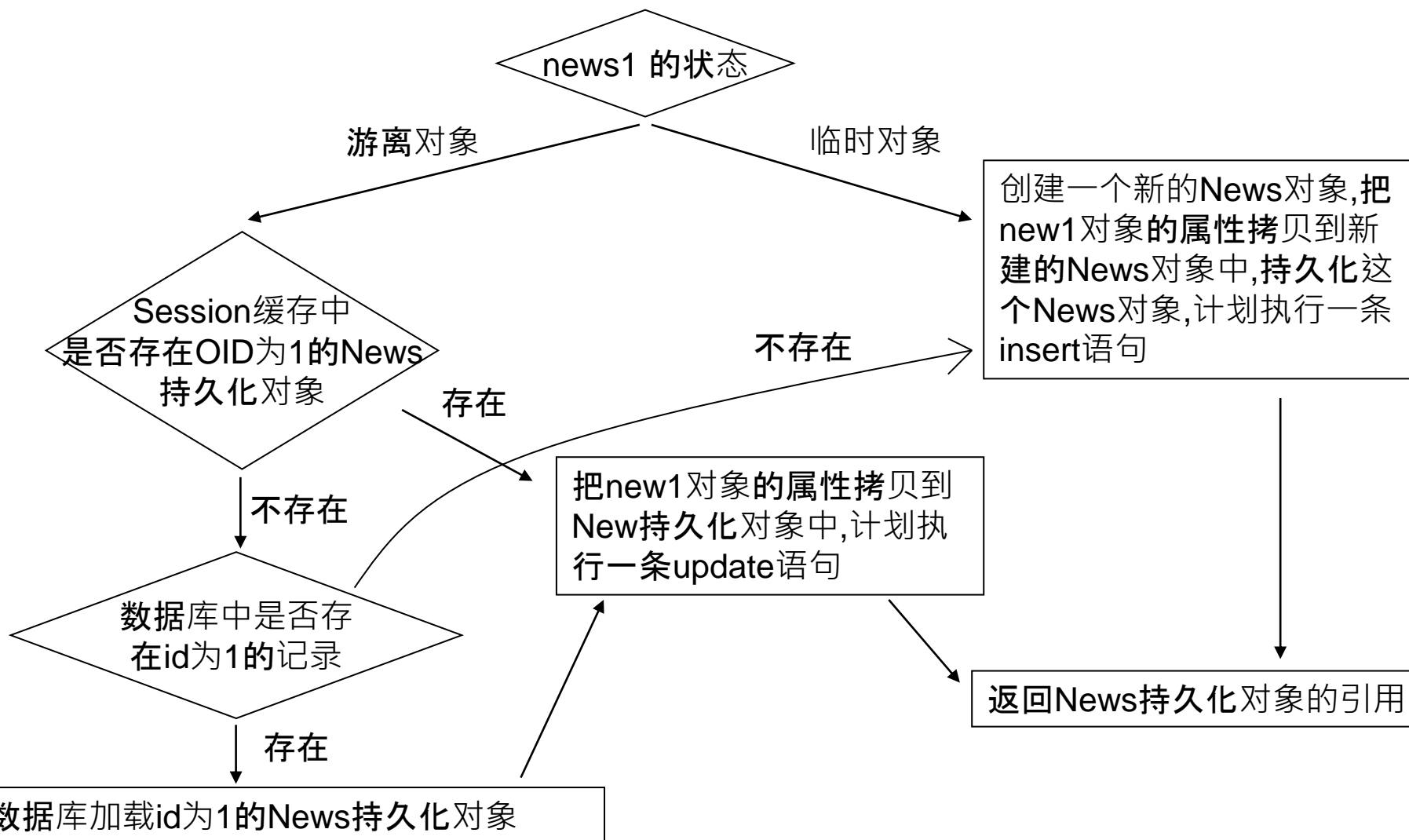
Session 的 saveOrUpdate() 方法

- 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 merge() 方法



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

通过 Hibernate 调用存储过程

- Work 接口: 直接通过 JDBC API 来访问数据库的操作

```
public interface Work {  
    //通过 JDBC API 来访问数据库的操作  
    public void execute(Connection connection) throws SQLException;  
}
```

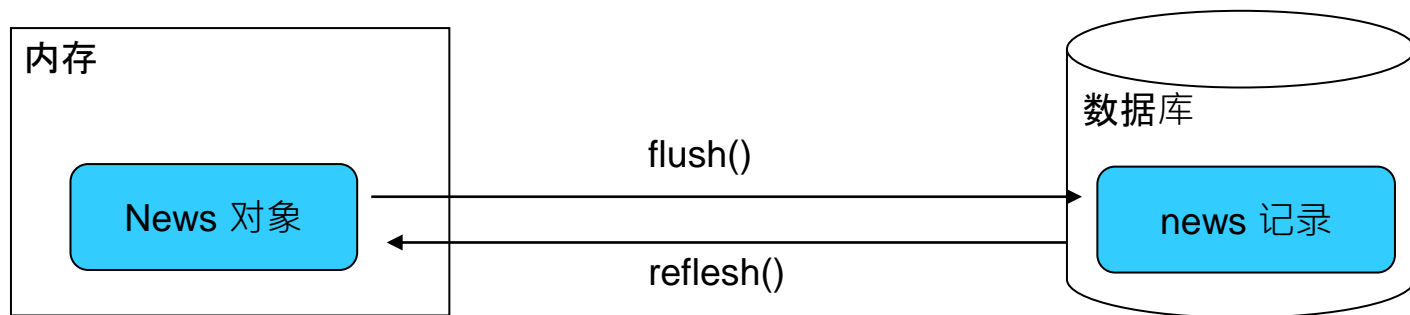
- Session 的 doWork(Work) 方法用于执行 Work 对象指定的操作, 即调用 Work 对象的 execute() 方法. Session 会把当前使用的数据库连接传递给 execute() 方法.

```
Work work = new Work(){  
    @Override  
    public void execute(Connection connection) throws SQLException {  
        String procedure = "{call testProcedure()}";  
        CallableStatement cstmt = connection.prepareCall(procedure);  
        cstmt.executeUpdate();  
    }  
};
```

```
session.doWork(work);
```

Hibernate 与触发器协同工作

- Hibernate 与数据库中的触发器协同工作时, 会造成两类问题
 - 触发器使 Session 的缓存中的持久化对象与数据库中对应的数据不一致: 触发器运行在数据库中, 它执行的操作对 Session 是透明的
 - Session 的 update() 方法盲目地激发触发器: 无论游离对象的属性是否发生变化, 都会执行 update 语句, 而 update 语句会激发数据库中相应的触发器
- 解决方案:
 - 在执行完 Session 的相关操作后, 立即调用 Session 的 flush() 和 refresh() 方法, 迫使 Session 的缓存与数据库同步(refresh() 方法重新从数据库中加载对象)



- 在映射文件的 <class> 元素中设置 select-before-update 属性: 当 Session 的 update 或 saveOrUpdate() 方法更新一个游离对象时, 会先执行 Select 语句, 获得当前游离对象在数据库中的最新数据, 只有在不一致的情况下才会执行 update 语句

Hibernate 的配置文件

讲师：佟刚

新浪微博：尚硅谷-佟刚

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

讲师：佟刚

新浪微博：尚硅谷-佟刚

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

- ③ auto-import="true"
- ③ catalog
- ③ default-access="property" ●
- ③ default-cascade="none" ●
- ③ default-lazy="true" ●
- ③ package ●
- ③ 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

⑧ abstract="true"
⑧ batch-size ●
⑧ catalog
⑧ check
⑧ discriminator-value ●
⑧ dynamic-insert="false" ●
⑧ dynamic-update="false" ●
⑧ entity-name
⑧ lazy="true" ●
⑧ mutable="true" ●
⑧ name ●
⑧ node
⑧ optimistic-lock="version"
⑧ persister
⑧ polymorphism="implicit"
⑧ proxy
⑧ rowid
⑧ schema
⑧ select-before-update="false" ●
⑧ subselect
⑧ table ●
⑧ 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

- 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

generator

• `@class` ●

- **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。

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

- 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.

⑧ name ●
⑧ access ●
⑧ column ●
⑧ formula
⑧ generated="never"
⑧ index
⑧ insert="true"
⑧ lazy="false" ●
⑧ length
⑧ node
⑧ not-null="true"
⑧ optimistic-lock="true"
⑧ precision
⑧ scale
⑧ type ●
⑧ unique-key
⑧ unique="false" ●
⑧ update="true" ●

Property

- ① name
- ① access
- ① column
- ① formula ● *
- ① generated="never"
- ① index ●
- ① insert="true"
- ① lazy="false" ●
- ① length ●
- ① node
- ① not-null="true"
- ① optimistic-lock="true"
- ① precision
- ① scale ●
- ① type
- ① unique-key
- ① unique="false" ●
- ① 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`.

Java 大对象类型的 Hiberante 映射

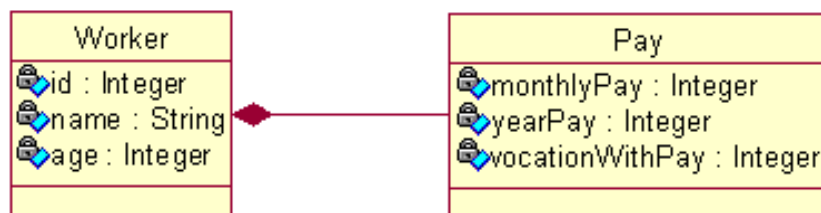
- 在 Java 中, `java.lang.String` 可用于表示长字符串(长度超过 255), 字节数组 `byte[]` 可用于存放图片或文件的二进制数据. 此外, 在 JDBC API 中还提供了 `java.sql.Clob` 和 `java.sql.Blob` 类型, 它们分别和标准 SQL 中的 CLOB 和 BLOB 类型对应. CLOB 表示字符串大对象 (Character Large Object), BLOB表示二进制对象(Binary Large Object)

映射类型	Java 类型	标准 SQL 类型	MYSQL 类型	Oracle 类型
binary	<code>byte[]</code>	VARCHAR(或BLOB)	BLOB	BLOB
text	<code>java.lang.String</code>	CLOB	TEXT	CLOB
clob	<code>java.sql.Clob</code>	CLOB	TEXT	CLOB
blob	<code>java.sql.Blob</code>	BLOB	BLOB	BLOB

- Mysql 不支持标准 SQL 的 CLOB 类型, 在 Mysql 中, 用 TEXT, MEDIUMTEXT 及 LONGTEXT 类型来表示长度操作 255 的长文本数据
- 在持久化类中, 二进制大对象可以声明为 `byte[]` 或 `java.sql.Blob` 类型; 字符串可以声明为 `java.lang.String` 或 `java.sql.Clob`
- 实际上在 Java 应用程序中处理长度超过 255 的字符串, 使用 `java.lang.String` 比 `java.sql.Clob` 更方便

映射组成关系

- 建立域模型和关系数据模型有着不同的出发点:
 - 域模型: 由程序代码组成, 通过细化持久化类的的粒度可提高代码的可重用性, 简化编程



- 在没有数据冗余的情况下, 应该尽可能减少表的数目, 简化表之间的参照关系, 以便提高数据的访问速度

主索引	ID		unique
ID	int(11)	no	<auto_increment>
NAME	varchar(30)	yes	<空>
AGE	int(11)	yes	<空>
MONTHLY_PAY	int(11)	yes	<空>
VOCATION_WITH_PAY	int(11)	yes	<空>
YEAR_PAY	int(11)	yes	<空>

映射组成关系

- Hibernate 把持久化类的属性分为两种:
 - 值(value)类型: **没有 OID, 不能被单独持久化, 生命周期依赖于所属的持久化类的对象的生命周期**
 - 实体(entity)类型: 有 OID, 可以被单独持久化, 有独立的生命周期
- 显然无法直接用 property 映射 pay 属性
- Hibernate 使用 <component> 元素来映射组成关系, 该元素表名 pay 属性是 Worker 类一个组成部分, 在 Hibernate 中称之为**组件**

```
<component name="pay" class="Pay">
    <parent name="worker" />

    <property name="monthlyPay" column="MONTHLY_PAY" type="integer"/>
    <property name="vocationWithPay" column="VOCATION_WITH_PAY" type="integer"/>
    <property name="yearPay" column="YEAR_PAY" type="integer" />
</component>
```

component

```
@name  
@access  
@class ●  
@insert="true"  
@lazy="false"  
@node  
@optimistic-lock="true"  
@unique="false"  
@update="true"
```

- **<component>** 元素来映射组成关系
 - **class**: 设定组成关系属性的类型, 此处表明 **pay** 属性为 **Pay** 类型

- **<parent>** 元素指定组件属性所属的整体类
 - **name**: 整体类在组件类中的属性名

```
@name
```

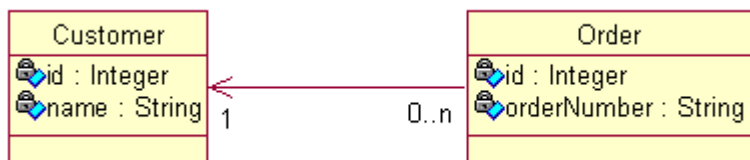

映射一对多关联关系

讲师：佟刚

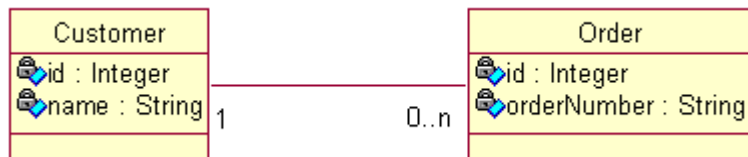
新浪微博：尚硅谷-佟刚

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

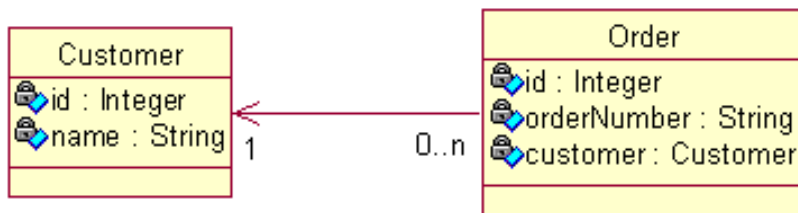


双向关联

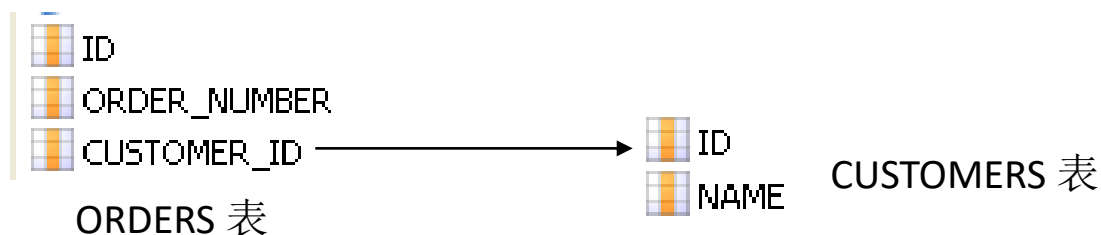


单向 n-1

- 单向 n-1 关联只需从 n 的一端可以访问 1 的一端
- 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"

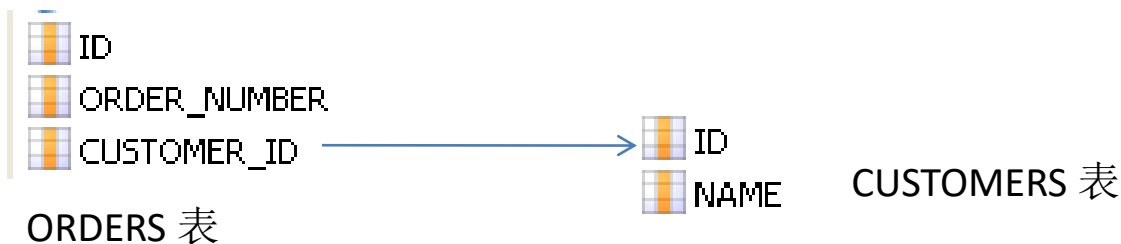
- **<many-to-one>** 元素来映射组成关系
 - **name**: 设定待映射的持久化类的属性的名字
 - **column**: 设定和持久化类的属性对应的表的外键
 - **class** : 设定待映射的持久化类的属性的类型

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

- **<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

- ① @name ●
- ① access
- ① batch-size ●
- ① cascade
- ① catalog
- ① check
- ① collection-type
- ① embed-xml="true"
- ① fetch="join" ●
- ① inverse="false" ●
- ① lazy="true" ●
- ① mutable="true"
- ① node
- ① optimistic-lock="true"
- ① order-by
- ① outer-join="true"
- ① persister
- ① schema
- ① sort="unsorted" ●
- ① subselect ●
- ① table
- ① where

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名字, 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

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

讲师：佟刚

新浪微博：尚硅谷-佟刚

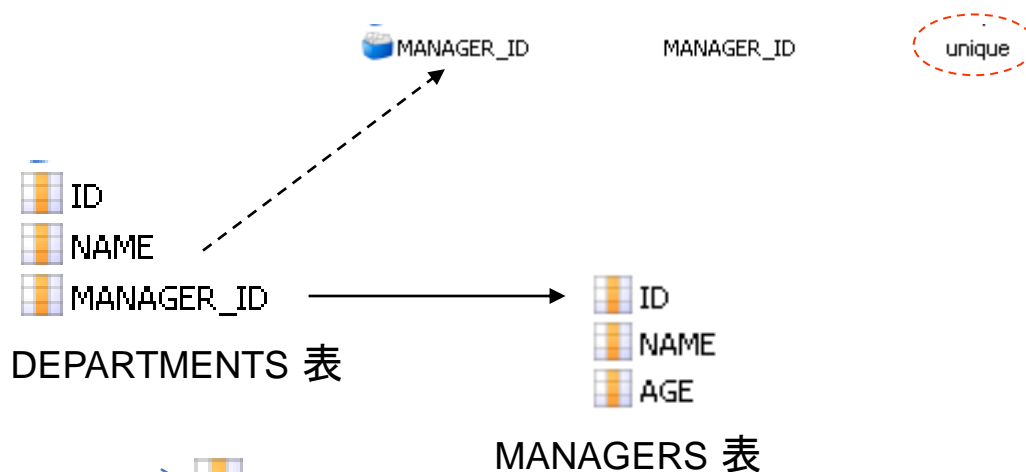
1 - 1

• 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

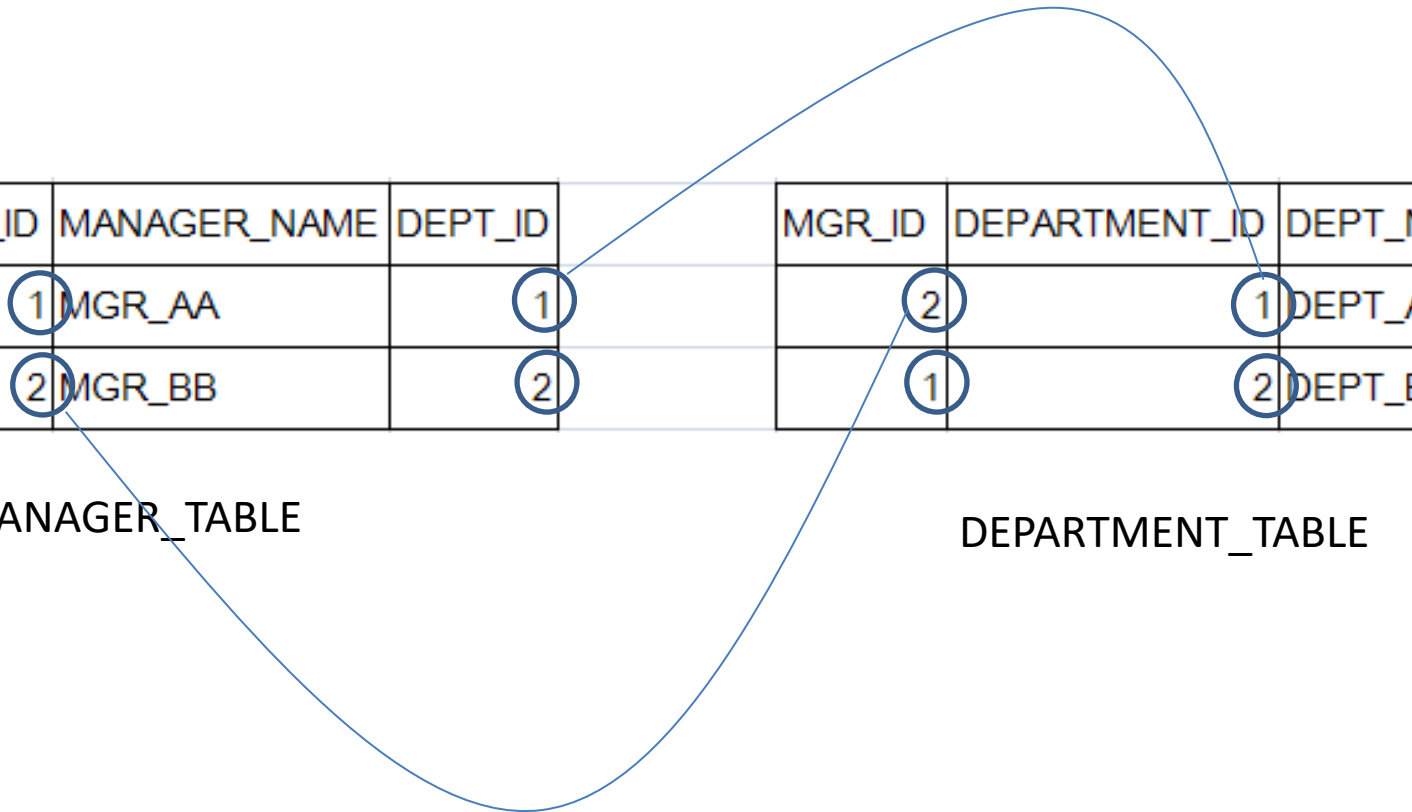
```
from MANAGERS manager0_
left outer join DEPARTMENTS department1_
on manager0_.ID=department1_.MANAGER_ID
where manager0_.ID=?
```

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"/>
```

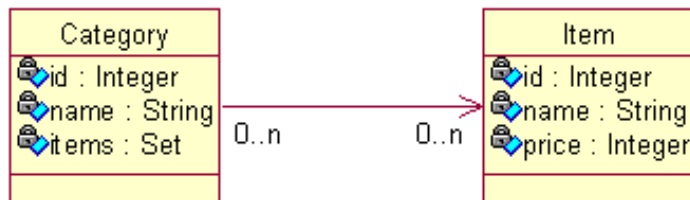

映射多对多关联关系

讲师：佟刚

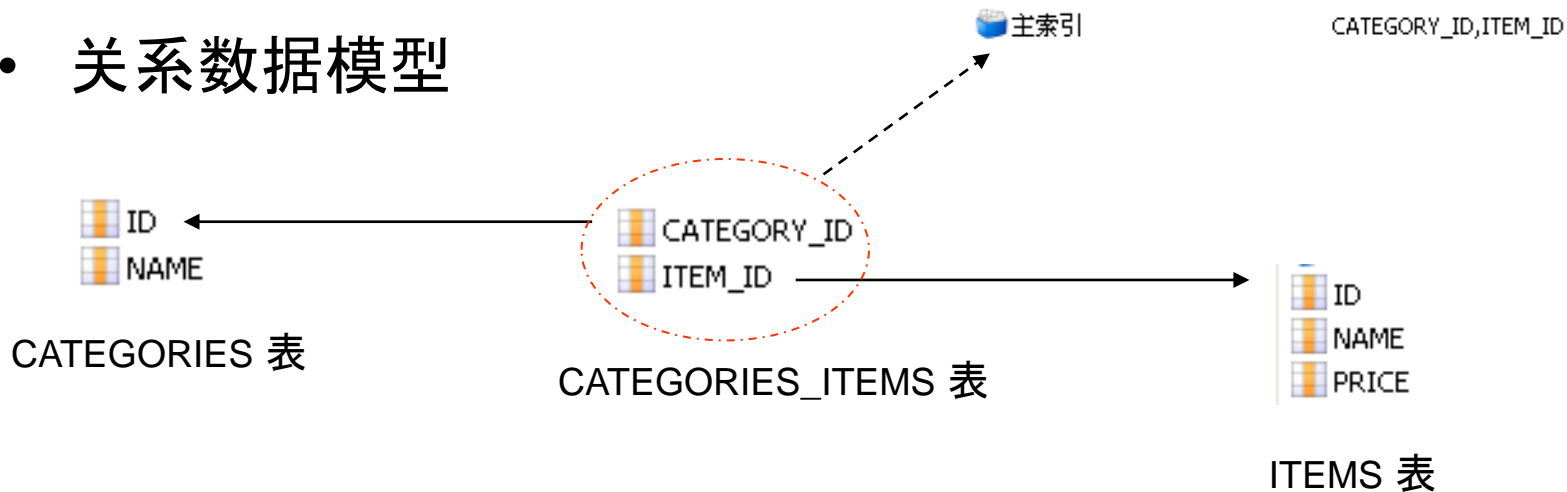
新浪微博：尚硅谷-佟刚

单向 n-n

- 域模型:



- 关系数据模型



NAME	ID
CATEGORY_AA	1
CATEGORY_BB	2
CATEGORY_CC	3
CATEGORY_DD	4

C_ID	ID	NAME
1	1	ITEM_AA
1	2	ITEM_BB
1	3	ITEM_CC
1	4	ITEM_DD

NAME	ID
CATEGORY_AA	1
CATEGORY_BB	2
CATEGORY_CC	3
CATEGORY_DD	4

C_ID	I_ID
1	1
1	2
1	3
1	4
2	1
2	2
2	3

ID	NAME
1	ITEM_AA
2	ITEM_BB
3	ITEM_CC
4	ITEM_DD

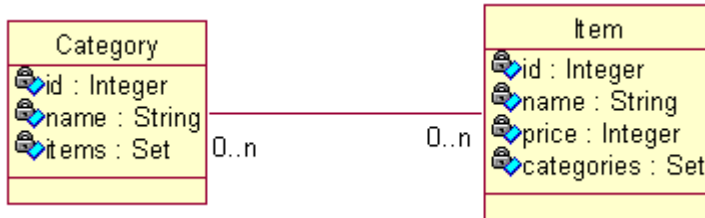
单向 n-n

- **n-n 的关联必须使用连接表**
- 与 1-n 映射类似，**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 关联映射不同的是，建立 n-n 关联时，集合中的元素使用 **many-to-many**. many-to-many 子元素的 class 属性指定 items 集合中存放的是 Item 对象，**column 属性指定 CATEGORIES_ITEMS 表中参照 ITEMS 表的外键为 ITEM_ID**

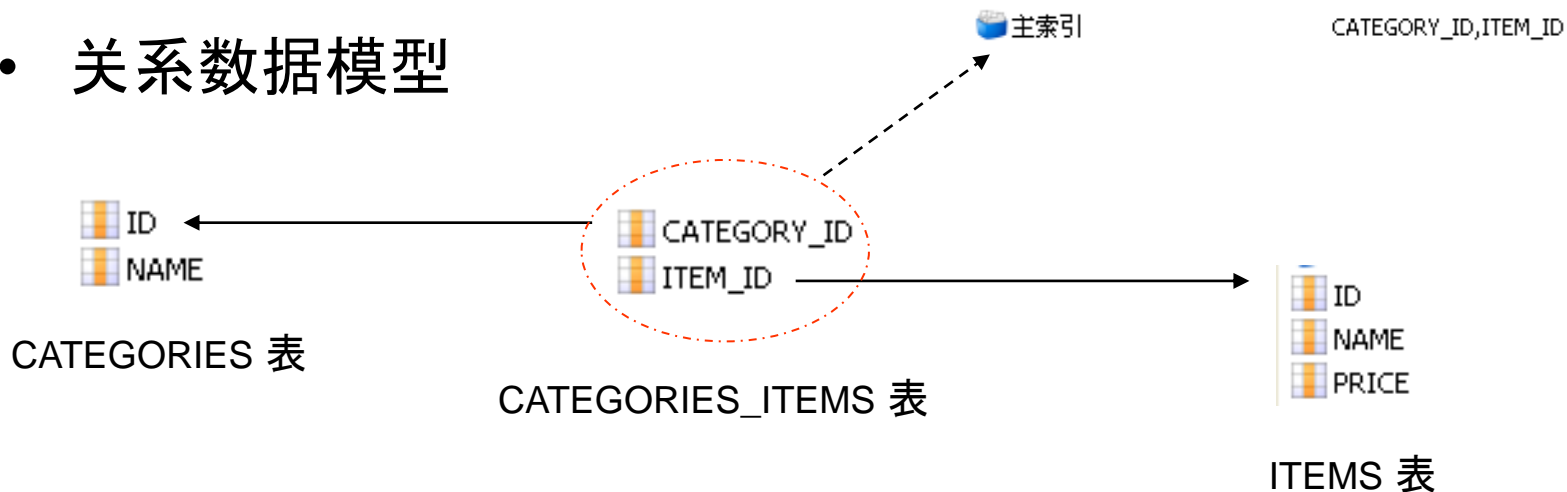
```
<set name="items" table="CATEGORIES_ITEMS" cascade="save-update">  
  <key column="CATEGORY_ID"></key>  
  <many-to-many class="Item" column="ITEM_ID"/>  
</set>
```

双向 n-n

- 域模型:



- 关系数据模型



双向n-n关联

- 双向 n-n 关联需要**两端都使用集合属性**
- 双向n-n关联**必须使用连接表**
- 集合属性应增加 key 子元素用以映射外键列, 集合元素里还应增加many-to-many子元素关联实体类
- **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>
```

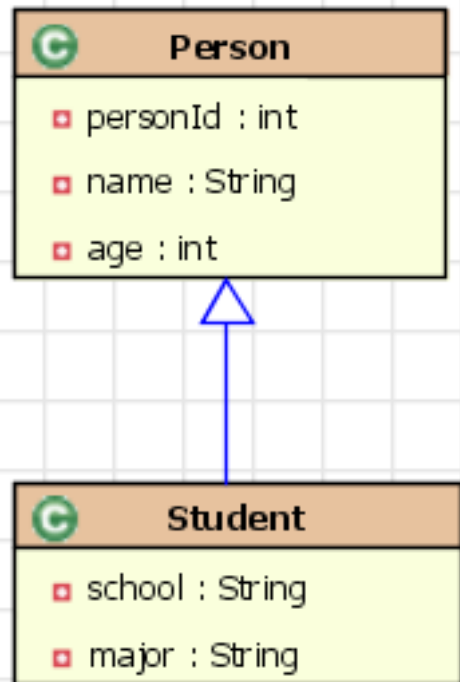

映射继承关系

讲师：佟刚

新浪微博：尚硅谷-佟刚

继承映射

- 对于面向对象的程序设计语言而言，继承和多态是两个最基本的概念。**Hibernate 的继承映射可以理解持久化类之间的继承关系**。例如：人和学生之间的关系。学生继承了人，可以认为学生是一个特殊的人，如果对人进行查询，学生的实例也将被得到。



继承映射

- Hibernate支持三种继承映射策略：
 - **使用 subclass 进行映射**：将域模型中的每一个实体对象映射到一个独立的表中，也就是说不用在关系数据模型中考虑域模型中的继承关系和多态。
 - **使用 joined-subclass 进行映射**：对于继承关系中的子类使用同一个表，这就需要在数据库表中增加额外的区分子类类型的字段。
 - **使用 union-subclass 进行映射**：域模型中的每个类映射到一个表，通过关系数据模型中的外键来描述表之间的继承关系。这也就相当于按照域模型的结构来建立数据库中的表，并通过外键来建立表之间的继承关系。

采用 subclass 元素的继承映射

- 采用 subclass 的继承映射可以实现对于继承关系中**父类和子类使用同一张表**
- 因为父类和子类的实例全部保存在同一个表中，因此**需要在该表内增加一列**，使用该列来区分每行记录到底是哪个类的实例----这个列被称为辨别者列(**discriminator**)
- 在这种映射策略下，**使用 subclass 来映射子类，使用 class 或 subclass 的 discriminator-value 属性指定辨别者列的值**
- **所有子类定义的字段都不能有非空约束**。如果为那些字段添加非空约束，那么父类的实例在那些列其实并没有值，这将引起数据库完整性冲突，导致父类的实例无法保存到数据库中

ID	TYPE	PERSON_NAME	AGE	SCHOOL
1	PERSON	AA	12	(NULL)
2	STUDENT	BB	13	ATGUIGU

辨别者列

子类独有的列

采用 subclass 元素的继承映射

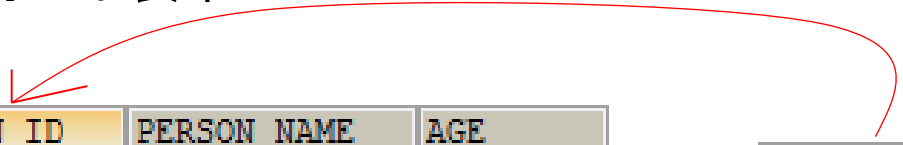
```
<class name="Person" table="person" discriminator-value="person">
  <id name="personId" column="person_id">
    <generator class="native"/>
  </id>

  <discriminator column="discriminator" type="string"/>
  <property name="name" length="30" />
  <property name="age" length="3" />

  <subclass name="Student" discriminator-value="student">
    <property name="school" length="30" />
    <property name="major" length="30" />
  </subclass>
</class>
```

采用 joined-subclass 元素的继承映射

- 采用 joined-subclass 元素的继承映射可以实现**每个子类一张表**
- 采用这种映射策略时，父类实例保存在父类表中，**子类实例由父类表和子类表共同存储**。因为子类实例也是一个特殊的父类实例，因此必然也包含了父类实例的属性。于是将子类和父类共有的属性保存在父类表中，子类增加的属性，则保存在子类表中。
- 在这种映射策略下，无须使用鉴别者列，但需要为每个子类使用 **key 元素映射共有主键**。
- **子类增加的属性可以添加非空约束**。因为子类的属性和父类的属性没有保存在同一个表中



PERSON_ID	PERSON_NAME	AGE
1	AA	12
2	BB	13

persons 表

STUDENT_ID	SCHOOL
2	ATGUIGU

students 表

采用 joined-subclass 元素的继承映射

```
<class name="Person" table="person">
  <id name="personId" column="person_id">
    <generator class="native"/>
  </id>

  <property name="name" length="30" />
  <property name="age" length="3" />

  <joined-subclass name="Student">
    <key column="person_id" not-null="true"></key>
    <property name="school" length="30" />
    <property name="major" length="30" />
  </joined-subclass>
</class>
```

采用 union-subclass 元素的继承映射

- 采用 union-subclass 元素可以实现**将每一个实体对象映射到一个独立的表中**。
- **子类增加的属性可以有非空约束** --- 即父类实例的数据保存在父表中, 而子类实例的数据保存在子类表中。
- **子类实例的数据仅保存在子类表中**, 而在父类表中没有任何记录
- 在这种映射策略下, 子类表的字段会比父类表的映射字段要多, 因为子类表的字段等于父类表的字段、加子类增加属性的总和
- 在这种映射策略下, **既不需要使用鉴别者列, 也无须使用 key 元素来映射共有主键**。
- **使用 union-subclass 映射策略是不可使用 identity 的主键生成策略**, 因为同一类继承层次中所有实体类都需要使用同一个主键种子, 即多个持久化实体对应的记录的主键应该是连续的. 受此影响, 也不该使用 native 主键生成策略, 因为 native 会根据数据库来选择使用 identity 或 sequence.

PERSON_ID	PERSON_NAME	AGE
1	AA	12

persons 表

PERSON_ID	PERSON_NAME	AGE	SCHOOL
2	BB	13	ATGUIGU

students 表

采用 union-subclass 元素的继承映射

```
<class name="Person" table="person">
  <id name="personId" column="person_id">
    <generator class="increment"/>
  </id>

  <property name="name" length="30" />
  <property name="age" length="3" />

  <u>union-subclass name="Student">
    <property name="school" length="30" />
    <property name="major" length="30" />
  </union-subclass>

</class>
```

三种继承映射方式的比较

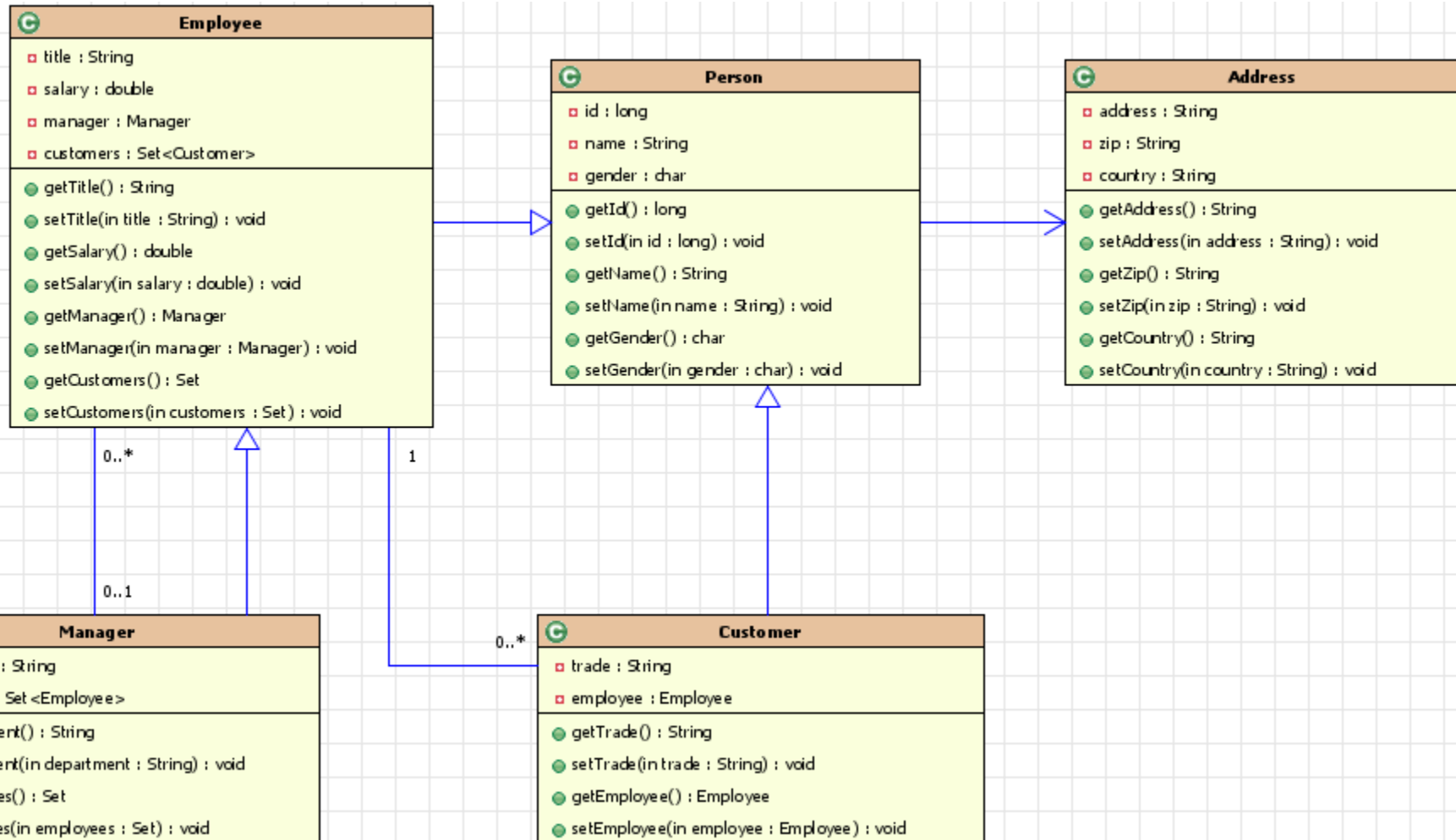
union-subclass

subclass

joined-subclass

比较方面	每个具体类一张表	每个类分层结构一张表	每个子类一张表
建立关系模型的原则	每个具体类对应 一张表，有多少具体类就需要建立多少个独立的表	描述一个继承关系只用一张表	每个子类使用一张表。但这些子类所对应的表都关联到基类所对应的表中
关系模型的优缺点	这种设计方式符合关系模型的设计原则，但有表中存在重复字段的问题	缺点有二：首先表中引入了区分子类的字段。其次，如果某个子类的某个属性的值不能为空，那么在数据库一级是不能设置该字段为NOT NULL的	这种设计方式完全符合关系模型的设计原则，而且不存在冗余
可维护性	如果需要对基类进行修改，则需要对基类以及该类的子类所对应的所有表都进行修改	维护起来比较方便，只需要修改一张表	维护起来比较方便，对每个类的修改只需要修改其所对应的表
灵活性	映射的灵活性很大，子类可以对包括基类属性在内的每一个属性进行单独的配置	灵活性差，表中的冗余字段会随着子类的增多而增加	灵活性很好，完全是参照对象继承的方式进行映射配置
查询的性能	对于子类的查询只需要访问单独的表，但对于父类的查询则需要检索所有的表	在任何情况下的查询都只需处理这一张表	对于父类的查询需要使用左外连接，而对于子类的查询则需要内连接
维护的性能	对于单个对象的持久化操作只需要处理一个表	对于单个对象的持久化操作只需处理一个表	对于子类的持久化操作至少需要处理两个表

示例代码



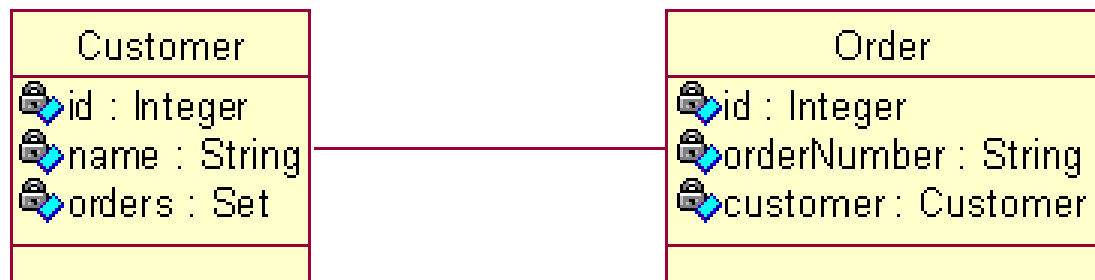
Hibernate 检索策略

讲师：佟刚

新浪微博：尚硅谷-佟刚

概述

- 检索数据时的 2 个问题：
 - 不浪费内存：当 Hibernate 从数据库中加载 Customer 对象时, 如果同时加载所有关联的 Order 对象, 而程序实际上仅仅需要访问 Customer 对象, 那么这些关联的 Order 对象就白白浪费了许多内存.
 - 更高的查询效率：发送尽可能少的 SQL 语句



类级别的检索策略

- 类级别可选的检索策略包括立即检索和延迟检索, 默认为延迟检索
 - 立即检索: 立即加载检索方法指定的对象
 - 延迟检索: 延迟加载检索方法指定的对象。在使用具体的属性时, 再进行加载
- 类级别的检索策略可以通过 **<class>** 元素的 **lazy** 属性进行设置
- 如果程序加载一个对象的目的是为了访问它的属性, 可以采取立即检索.
- 如果程序加载一个持久化对象的目的是仅仅为了获得它的引用, 可以采用延迟检索。注意出现懒加载异常!

类级别的检索策略

- 无论 `<class>` 元素的 `lazy` 属性是 `true` 还是 `false`, `Session` 的 `get()` 方法及 `Query` 的 `list()` 方法在类级别总是使用立即检索策略
- 若 `<class>` 元素的 `lazy` 属性为 `true` 或取默认值, `Session` 的 `load()` 方法不会执行查询数据表的 `SELECT` 语句, 仅返回代理类对象的实例, 该代理类实例有如下特征:
 - 由 `Hibernate` 在运行时采用 `CGLIB` 工具动态生成
 - `Hibernate` 创建代理类实例时, 仅初始化其 **OID 属性**
 - 在应用程序第一次访问代理类实例的非 `OID` 属性时, `Hibernate` 会初始化代理类实例

一对多和多对多的检索策略

- 在映射文件中, 用 `<set>` 元素来配置一对多关联及多对多关联关系. `<set>` 元素有 `lazy` 和 `fetch` 属性
 - **lazy: 主要决定 orders 集合被初始化的时机.** 即到底是在加载 Customer 对象时就被初始化, 还是在程序访问 orders 集合时被初始化
 - **fetch: 取值为 “select” 或 “subselect” 时, 决定初始化 orders 的查询语句的形式; 若取值为 “join”, 则决定 orders 集合被初始化的时机**
 - **若把 fetch 设置为 “join”, lazy 属性将被忽略**
 - `<set>` 元素的 `batch-size` 属性: 用来为延迟检索策略或立即检索策略设定批量检索的数量. 批量检索能减少 SELECT 语句的数目, 提高延迟检索或立即检索的运行性能.

<set> 元素的 lazy 和 fetch 属性

lazy 属性 (默认值为 true)	fetch 属性 (默认值为 select)	检索策略
true	未显式设置 (取默认值 select)	采用延迟检索策略, 这是默认的检索策略, 也是有限考虑使用的检索策略
false	未显式设置 (取默认值 select)	采用立即检索策略, 当使用 Hibernate 二级缓存时, 可以考虑使用立即检索
extra	未显式设置 (取默认值 select)	采用加强延迟检索策略, 它尽可能延迟 orders 集合被初始化的时机
true, false 或 extra	未显式设置 (取默认值 select)	lazy 属性决定采用的检索策略, 即决定初始化 orders 集合的时机. fetch 属性为 select , 意味着通过 select 语句来初始化 orders 集合, 形式为 SELECT * FROM orders WHERE customer_id IN (1, 2, 3, 4)
true, false 或 extra	subselect	lazy 属性决定采用的检索策略, 即决定初始化 orders 集合的时机. fetch 属性为 subselect , 意味着通过 subselect 语句来初始化 orders 集合, 形式为 SELECT * FROM orders WHERE customer_id IN (SELECT id FROM customers)
未显式设置 (取默认值 true)	join	采用迫切左外连接策略

延迟检索和增强延迟检索

- 在延迟检索(lazy 属性值为 true) 集合属性时, Hibernate 在以下情况下初始化集合代理类实例
 - 应用程序第一次访问集合属性: iterator(), size(), isEmpty(), contains() 等方法
 - 通过 Hibernate.initialize() 静态方法显式初始化
- 增强延迟检索(lazy 属性为 extra): 与 lazy="true" 类似. 主要区别是**增强延迟检索策略能进一步延迟 Customer 对象的 orders 集合代理实例的初始化时机** :
 - 当程序第一次访问 orders 属性的 iterator() 方法时, 会导致 orders 集合代理类实例的初始化
 - 当程序第一次访问 order 属性的 size(), contains() 和 isEmpty() 方法时, Hibernate 不会初始化 orders 集合类的实例, 仅通过特定的 select 语句查询必要的信息, 不会检索所有的 Order 对象

<set> 元素的 batch-size 属性

- <set> 元素有一个 batch-size 属性, 用来为延迟检索策略或立即检索策略设定批量检索的数量. 批量检索能减少 SELECT 语句的数目, 提高延迟检索或立即检索的运行性能.

一对多和多对多的检索策略

- 在映射文件中, 用 `<set>` 元素来配置一对多关联及多对多关联关系. `<set>` 元素有 `lazy` 和 `fetch` 属性
 - **lazy: 主要决定 orders 集合被初始化的时机.** 即到底是在加载 Customer 对象时就被初始化, 还是在程序访问 orders 集合时被初始化
 - **fetch: 取值为 “select” 或 “subselect” 时, 决定初始化 orders 的查询语句的形式; 若取值为 “join”, 则决定 orders 集合被初始化的时机**
 - 若把 `fetch` 设置为 “join”, `lazy` 属性将被忽略

用带子查询的 select 语句整批量初始化 orders 集合(fetch 属性为 “subselect”)

- <set> 元素的 fetch 属性: 取值为 “select” 或 “subselect” 时, 决定初始化 orders 的查询语句的形式; 若取值为 “join”, 则决定 orders 集合被初始化的时机. 默认值为 select
- 当 fetch 属性为 “subselect” 时
 - 假定 Session 缓存中有 n 个 orders 集合代理类实例没有被初始化, Hibernate 能够通过带子查询的 select 语句, 来批量初始化 n 个 orders 集合代理类实例
 - batch-size 属性将被忽略
 - 子查询中的 select 语句为查询 CUSTOMERS 表 OID 的 SELECT 语句

迫切左外连接检索(fetch 属性值设为 “join”)

- <set> 元素的 fetch 属性: 取值为 “select” 或 “subselect” 时, 决定初始化 orders 的查询语句的形式; **若取值为 “join”, 则决定 orders 集合被初始化的时机**. 默认值为 select
- 当 fetch 属性为 “join” 时:
 - 检索 Customer 对象时, 会采用**迫切左外连接**(通过左外连接加载与检索指定的对象关联的对象)策略来检索所有关联的 Order 对象
 - **lazy 属性将被忽略**
 - **Query 的 list() 方法会忽略映射文件中配置的迫切左外连接检索策略, 而依旧采用延迟加载策略**

多对一和一对一关联的检索策略

- 和 <set> 一样, <many-to-one> 元素也有一个 lazy 属性和 fetch 属性.

lazy 属性 (默认值为 proxy)	fetch 属性 (默认值为 select)	检索 Order 对象时对关联的 Customer 对象使用的检索策略
proxy	未显式设置 (取默认值 select)	采用延迟检索
no-proxy	未显式设置 (取默认值 select)	无代理延迟检索
FALSE	未显式设置 (取默认值 select)	立即检索
未显式设置 (取默认值 proxy)	join	迫切左外连接策略

- 若 fetch 属性设为 join, 那么 lazy 属性被忽略
- 迫切左外连接检索策略的优点在于比立即检索策略使用的 SELECT 语句更少.
- 无代理延迟检索需要增强持久化类的字节码才能实现

多对一和一对一关联的检索策略

- Query 的 list 方法会忽略映射文件配置的迫切左外连接检索策略, 而采用延迟检索策略
- 如果在关联级别使用了延迟加载或立即加载检索策略, 可以**设定批量检索的大小**, 以帮助提高延迟检索或立即检索的运行性能.
- Hibernate 允许在应用程序中覆盖映射文件中设定的检索策略.

检索策略小结

- 类级别和关联级别可选的检索策略及默认的检索策略

检索策略的作用域	可选的检索策略	默认的检索策略	运行时行为受影响的检索方法
类级别	立即检索 延迟检索	延迟检索	仅影响 Session 的 load() 方法
关联级别	立即检索 延迟检索 迫切左外连接检索	延迟检索	影响 Session 的 load() 和 get() 方法, 以及 Query API 和 Criteria API; 例外情况是 Query API 会忽略映射文件中设定的迫切左外连接检索策略

- 3 种检索策略的运行机制

检索策略的类型	类级别	关联级别
立即检索	立即加载检索方法指定的对象	立即加载与检索方法指定的对象的关联对象, 可以设定批量检索数量
延迟检索	延迟加载检索方法指定的对象	延迟加载与检索方法指定的对象的关联对象, 可以设定批量检索数量
迫切左外连接检索	不适用	通过左外连接加载与检索方法指定的对象的关联对象

检索策略小结

- 映射文件中用于设定检索策略的几个属性

属性	类级别	一对多关联级别	多对多关联级别
lazy	1.<class> 元素中lazy属性的可选值为: true(延迟检索)和false(立即检索); 2.<class>元素的lazy属性的默认值为true	1.<set>元素中lazy属性的可选值为: true(延迟检索)、extra(增强延迟检索)和false(立即检索); 2.<set> 元素的lazy属性的默认值为true	1.<many-to-one>元素中lazy属性的可选值为: proxy(延迟检索)、no-proxy(无代理延迟检索)和false(立即检索); 2.<many-to-one>元素的lazy属性的默认值为true
fetch	没有此属性	1.<set>元素中fetch属性的可选值为: select(select查询语句)、subselect(带子查询的select查询语句)和join(迫切左外连接检索); 2.<set>元素的fetch属性的默认值为select	1.<many-to-one>元素中fetch属性的可选值为: select(select查询语句)和join(迫切左外连接检索); 2.<many-to-one>元素的fetch属性的默认值为select
batch-size	设定批量检索的数量。可选值为一个正整数, 默认值为1。如果设定此项, 合理的取值在3-10之间。仅适用于关联级别的立即检索和延迟检索。在<class>和<set>元素中包含此属性。		

检索策略小结

- 比较 Hibernate 的三种检索策略

检索策略	优点	缺点	优先考虑使用的场合
立即检索	对应用程序完全透明，不管对象处于持久化状态，还是游离状态，应用程序都可以方便地从一个对象导航到与它关联的对象	(1) select语句数目多 (2) 可能会加载应用程序不需要访问的对象，白白浪费许多内存空间	(1) 类级别 (2) 应用程序需要立即访问的对象 (3) 使用了第二级缓存
延迟检索	由应用程序决定需要加载那些对象，可以避免执行多余的select语句，以及避免加载应用程序不需要访问的对象。因此能提高检索性能，并且节省内在空间	应用程序如果希望访问游离状态的代理类实例，必须保证它在持久化状态时已经被初始化	(1) 一对多或者多对多关联 (2) 应用程序不需要立即访问或者根本不会访问的对象
迫切左外连接检索	(1) 对应用程序完全透明，不管对象处于持久化状态，还是游离状态，应用程序都可以方便地从一个对象导航到与它关联的对象 (2) 使用了外连接，select语句数目少	(1) 可能会加载应用程序不需要访问的对象，白白浪费许多内存空间 (2) 复杂的数据库表连接也会影响检索性能	(1) 多对一或者多对多关联 (2) 应用程序需要立即访问的对象 (3) 数据库系统具有良好的表连接性能

Hibernate 检索方式

讲师：佟刚

新浪微博：尚硅谷-佟刚

概述

- Hibernate 提供了以下几种检索对象的方式
 - **导航对象图检索方式**: 根据已经加载的对象导航到其他对象
 - **OID 检索方式**: 按照对象的 OID 来检索对象
 - **HQL 检索方式**: 使用面向对象的 HQL 查询语言
 - **QBC 检索方式**: 使用 QBC(Query By Criteria) API 来检索对象. 这种 API 封装了基于字符串形式的查询语句, 提供了更加面向对象的查询接口.
 - **本地 SQL 检索方式**: 使用本地数据库的 SQL 查询语句

HQL 检索方式

- HQL(Hibernate Query Language) 是面向对象的查询语言, 它和 SQL 查询语言有些相似. 在 Hibernate 提供的各种检索方式中, HQL 是使用最广的一种检索方式. 它有如下功能:
 - 在查询语句中设定各种查询条件
 - 支持投影查询, 即仅检索出对象的部分属性
 - 支持分页查询
 - 支持连接查询
 - 支持分组查询, 允许使用 HAVING 和 GROUP BY 关键字
 - 提供内置聚集函数, 如 sum(), min() 和 max()
 - 支持子查询
 - 支持动态绑定参数
 - 能够调用 用户定义的 SQL 函数或标准的 SQL 函数

HQL 检索方式

- HQL 检索方式包括以下步骤:
 - 通过 Session 的 createQuery() 方法创建一个 Query 对象, 它包括一个 HQL 查询语句. HQL 查询语句中可以包含命名参数
 - 动态绑定参数
 - 调用 Query 相关方法执行查询语句.
- **Query 接口支持方法链编程风格**, 它的 setXxx() 方法返回自身实例, 而不是 void 类型
- HQL vs SQL:
 - **HQL 查询语句是面向对象的, Hibernate 负责解析 HQL 查询语句**, 然后根据对象-关系映射文件中的映射信息, 把 HQL 查询语句**翻译**成相应的 SQL 语句. HQL 查询语句中的主体是**域模型中的类及类的属性**
 - SQL 查询语句是与关系数据库绑定在一起的. SQL 查询语句中的主体是数据库表及表的字段.

HQL 检索方式

- 绑定参数:
 - Hibernate 的参数绑定机制依赖于 JDBC API 中的 PreparedStatement 的预定义 SQL 语句功能.
 - HQL 的参数绑定由两种形式:
 - **按参数名字绑定**: 在 HQL 查询语句中定义命名参数, 命名参数以 “:” 开头.
 - 按参数位置绑定: 在 HQL 查询语句中用 “?” 来定义参数位置
 - 相关方法:
 - setEntity(): 把参数与一个持久化类绑定
 - setParameter(): 绑定任意类型的参数. 该方法的第三个参数显式指定 Hibernate 映射类型
- HQL 采用 **ORDER BY** 关键字对查询结果**排序**

HQL 检索方式

- 分页查询:

- **setFirstResult**(int firstResult): 设定从哪一个对象开始检索, 参数 firstResult 表示这个对象在查询结果中的索引位置, 索引位置的起始值为 0. 默认情况下, Query 从查询结果中的第一个对象开始检索
- **setMaxResults**(int maxResults): 设定一次最多检索出的对象的数目. 在默认情况下, Query 和 Criteria 接口检索出查询结果中所有的对象

HQL 检索方式

- 在映射文件中定义命名查询语句
 - Hibernate 允许在映射文件中定义字符串形式的查询语句.
 - **<query>** 元素用于定义一个 HQL 查询语句, 它和 **<class>** 元素并列.

```
<query name="findNewsByTitle">
  <![CDATA[
    FROM News n WHERE n.title LIKE :title
  ]]>
</query>
```
 - 在程序中通过 Session 的 `getNamedQuery()` 方法获取查询语句对应的 Query 对象.

投影查询

- 投影查询: 查询结果仅包含实体的部分属性. 通过 SELECT 关键字实现.
- Query 的 list() 方法返回的集合中包含的是数组类型的元素, 每个对象数组代表查询结果的一条记录
- 可以在持久化类中定义一个对象的构造器来包装投影查询返回的记录, 使程序代码能完全运用面向对象的语义来访问查询结果集.
- 可以通过 DISTINCT 关键字来保证查询结果不会返回重复元素

报表查询

- 报表查询用于对数据分组和统计, 与 SQL 一样, HQL 利用 **GROUP BY** 关键字对数据分组, 用 **HAVING** 关键字对分组数据设定约束条件.
- 在 HQL 查询语句中可以调用以下聚集函数
 - count()
 - min()
 - max()
 - sum()
 - avg()

HQL (迫切)左外连接

- 迫切左外连接:

- **LEFT JOIN FETCH** 关键字表示迫切左外连接检索策略.
- **list()** 方法返回的集合中存放实体对象的引用, 每个 Department 对象关联的 Employee 集合都被初始化, 存放所有关联的 Employee 的实体对象.
- 查询结果中可能会包含重复元素, 可以通过一个 HashSet 来过滤重复元素

- 左外连接:

- **LEFT JOIN** 关键字表示左外连接查询.
- **list()** 方法返回的集合中存放的是对象数组类型
- 根据配置文件来决定 Employee 集合的检索策略.
- 如果希望 **list()** 方法返回的集合中仅包含 Department 对象, 可以在 HQL 查询语句中使用 **SELECT** 关键字

HQL (迫切)内连接

- 迫切内连接:

- **INNER JOIN FETCH** 关键字表示迫切内连接, 也可以省略 INNER 关键字
- list() 方法返回的集合中存放 Department 对象的引用, 每个 Department 对象的 Employee 集合都被初始化, 存放所有关联的 Employee 对象

- 内连接:

- INNER JOIN 关键字表示内连接, 也可以省略 INNER 关键字
- list() 方法的集合中存放的每个元素对应查询结果的一条记录, 每个元素都是对象数组类型
- 如果希望 list() 方法的返回的集合仅包含 Department 对象, 可以在 HQL 查询语句中使用 SELECT 关键字

关联级别运行时的检索策略

- 如果在 HQL 中没有显式指定检索策略, 将使用映射文件配置的检索策略.
- HQL 会忽略映射文件中设置的迫切左外连接检索策略, **如果希望 HQL 采用迫切左外连接策略, 就必须在 HQL 查询语句中显式的指定它**
- 若在 HQL 代码中显式指定了检索策略, 就会覆盖映射文件中配置的检索策略

QBC 检索和本地 SQL 检索

- QBC 查询就是通过使用 Hibernate 提供的 Query By Criteria API 来查询对象，这种 API 封装了 SQL 语句的动态拼装，对查询提供了更加面向对象的功能接口
- 本地SQL查询来完善HQL不能涵盖所有的查询特性

Hibernate 二级缓存

讲师：佟刚

新浪微博：尚硅谷-佟刚

Hibernate 缓存

- 缓存(Cache): 计算机领域非常通用的概念。它**介于应用程序和永久性数据存储源(如硬盘上的文件或者数据库)之间**，其作用是**降低应用程序直接读写永久性数据存储源的频率，从而提高应用的运行性能**。缓存中的数据是数据存储源中数据的拷贝。**缓存的物理介质通常是内存**
- Hibernate中提供了两个级别的缓存
 - 第一级别的缓存是 Session 级别的缓存，它是属于事务范围的缓存。这一级别的缓存由 hibernate 管理的
 - 第二级别的缓存是 SessionFactory 级别的缓存，它是属于进程范围的缓存

SessionFactory 级别的缓存

- SessionFactory 的缓存可以分为两类：
 - 内置缓存: **Hibernate 自带的, 不可卸载**. 通常在 Hibernate 的初始化阶段, Hibernate 会把映射元数据和预定义的 SQL 语句放到 SessionFactory 的缓存中, 映射元数据是映射文件中数据 (.hbm.xml 文件中的数据) 的复制. 该内置缓存是只读的.
 - **外置缓存(二级缓存): 一个可配置的缓存插件**. 在默认情况下, SessionFactory 不会启用这个缓存插件. 外置缓存中的数据是数据库数据的复制, 外置缓存的物理介质可以是内存或硬盘

使用 Hibernate 的二级缓存

- 适合放入二级缓存中的数据：
 - 很少被修改
 - 不是很重要的数据, 允许出现偶尔的并发问题
- 不适合放入二级缓存中的数据：
 - 经常被修改
 - 财务数据, 绝对不允许出现并发问题
 - 与其他应用程序共享的数据

Hibernate 二级缓存的架构

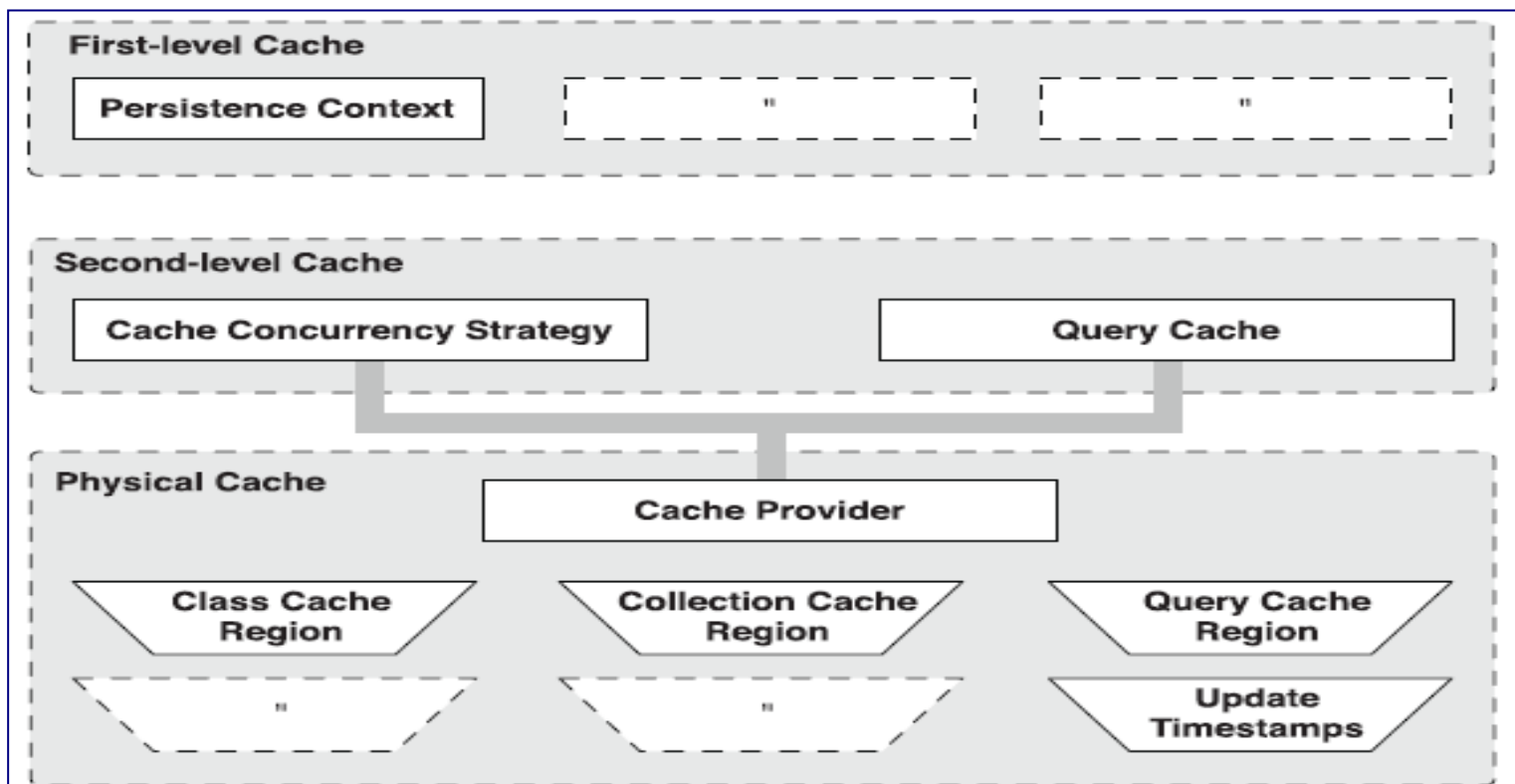


Figure 13.7 Hibernate's two-level cache architecture

二级缓存的并发访问策略

- 两个并发的事务同时访问持久层的缓存的相同数据时, 也有可能出现各类并发问题.
- 二级缓存可以设定以下 4 种类型的并发访问策略, 每一种访问策略对应一种事务隔离级别
 - 非严格读写(Nonstrict-read-write): 不保证缓存与数据库中数据的一致性. 提供 **Read Uncommitted 事务隔离级别**, 对于极少被修改, 而且允许脏读的数据, 可以采用这种策略
 - 读写型(Read-write): 提供 **Read Committed 数据隔离级别**. 对于经常读但是很少被修改的数据, 可以采用这种隔离类型, 因为它可以防止脏读
 - 事务型(Transaction): 仅在受管理环境下适用. 它提供了 **Repeatable Read 事务隔离级别**. 对于经常读但是很少被修改的数据, 可以采用这种隔离类型, 因为它可以防止脏读和不可重复读
 - 只读型(Read-Only): 提供 **Serializable 数据隔离级别**, 对于从来不会被修改的数据, 可以采用这种访问策略

管理 Hibernate 的二级缓存

- Hibernate 的二级缓存是进程或集群范围内的缓存
- 二级缓存是可配置的的插件, Hibernate 允许选用以下类型的缓存插件:
 - **EHCache**: 可作为进程范围内的缓存, 存放数据的物理介质可以使内存或硬盘, 对 Hibernate 的查询缓存提供了支持
 - OpenSymphony OSCache: 可作为进程范围内的缓存, 存放数据的物理介质可以使内存或硬盘, 提供了丰富的缓存数据过期策略, 对 Hibernate 的查询缓存提供了支持
 - SwarmCache: 可作为集群范围内的缓存, 但不支持 Hibernate 的查询缓存
 - JBossCache: 可作为集群范围内的缓存, 支持 Hibernate 的查询缓存
- 4 种缓存插件支持的并发访问策略(x 代表支持, 空白代表不支持)

Table 13.1 Cache concurrency strategy support

Concurrency strategy cache provider	Read-only	Nonstrict- read-write	Read-write	Transactional
EHCache	X	X	X	
OSCache	X	X	X	
SwarmCache	X	X		
JBoss Cache	X			X

配置进程范围内的二级缓存

- 配置进程范围内的二级缓存的步骤:
 - 选择合适的缓存插件: EHCACHE(jar 包和 配置文件), 并编译器配置文件
 - 在 Hibernate 的配置文件中启用二级缓存并指定和 EHCACHE 对应的缓存适配器
 - 选择需要使用二级缓存的持久化类, 设置它的二级缓存的并发访问策略
 - `<class>` 元素的 `cache` 子元素表明 Hibernate 会缓存对象的简单属性, 但不会缓存集合属性, 若希望缓存集合属性中的元素, 必须在 `<set>` 元素中加入 `<cache>` 子元素
 - 在 hibernate 配置文件中通过 `<class-cache/>` 节点配置使用缓存

ehcache.xml

- `<diskStore>`: 指定一个目录：当 EHCache 把数据写到硬盘上时, 将把数据写到这个目录下.
- `<defaultCache>`: 设置缓存的默认 **数据过期策略**
- `<cache>` 设定具体的 **命名缓存**的数据过期策略。 **每个命名缓存代表一个缓存区域**
- 缓存区域(region)：一个具有名称的缓存块，可以给每一个缓存块设置不同的缓存策略。如果没有设置任何的缓存区域，则所有被缓存的对象，都将使用默认的缓存策略。即：`<defaultCache.../>`
- Hibernate在不同的缓存区域保存不同的类/集合。
 - 对于类而言，区域的名称是类名。如：`com.atguigu.domain.Customer`
 - 对于集合而言，区域的名称是类名加属性名。如
`com.atguigu.domain.Customer.orders`

ehcache.xml

- cache 元素的属性

- name:设置缓存的名字,它的取值为类的全限定名或类的集合的名字
- maxInMemory:设置基于内存的缓存中可存放的对象最大数目
- eternal:设置对象是否为永久的,true表示永不过期,此时将忽略 timeToldleSeconds 和 timeToLiveSeconds属性; 默认值是false
- timeToldleSeconds:设置对象空闲最长时间,以秒为单位,超过这个时间,对象过期。当对象过期时,EHCache会把它从缓存中清除。如果此值为0,表示对象可以无限期地处于空闲状态。
- timeToLiveSeconds:设置对象生存最长时间,超过这个时间,对象过期。如果此值为0,表示对象可以无限期地存在于缓存中. 该属性值必须大于或等于 timeToldleSeconds 属性值
- overflowToDisk:设置基于内存的缓存中的对象数目达到上限后,是否把溢出的对象写到基于硬盘的缓存中

查询缓存

- 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查询缓存中. 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
- 查询缓存使用于如下场合:
 - 应用程序运行时经常使用查询语句
 - 很少对与查询语句检索到的数据进行插入, 删除和更新操作
- 启用查询缓存的步骤
 - 配置二级缓存, 因为查询缓存依赖于二级缓存
 - 在 hibernate 配置文件中启用查询缓存
 - 对于希望启用查询缓存的查询语句, 调用 Query 的 setCacheable() 方法

时间戳缓存区域

- 时间戳缓存区域存放了对于查询结果相关的表进行插入, 更新或删除操作的时间戳. **Hibernate 通过时间戳缓存区域来判断被缓存的查询结果是否过期, 其运行过程如下:**
 - T1 时刻执行查询操作, 把查询结果存放在 QueryCache 区域, 记录该区域的时间戳为 T1
 - T2 时刻对查询结果相关的表进行更新操作, Hibernate 把 T2 时刻存放在 UpdateTimestampCache 区域.
 - T3 时刻执行查询结果前, 先比较 QueryCache 区域的时间戳和 UpdateTimestampCache 区域的时间戳, 若 $T2 > T1$, 那么就丢弃原先存放在 QueryCache 区域的查询结果, 重新到数据库中查询数据, 再把结果存放到 QueryCache 区域; 若 $T2 < T1$, 直接从 QueryCache 中获得查询结果

Query 接口的 iterate() 方法

- Query 接口的 iterator() 方法
 - 同 list() 一样也能执行查询操作
 - list() 方法执行的 SQL 语句包含实体类对应的数据表的所有字段
 - Iterator() 方法执行的 SQL 语句中**仅包含实体类对应的数据表的 ID 字段**
 - **当遍历访问结果集时, 该方法先到 Session 缓存及二级缓存中查看是否存在特定 OID 的对象, 如果存在, 就直接返回该对象, 如果不存在该对象就通过相应的 SQL Select 语句到数据库中加载特定的实体对象**
- 大多数情况下, 应考虑使用 list() 方法执行查询操作.
iterator() 方法仅在满足以下条件的场合, 可以**稍微**提高查询性能:
 - 要查询的数据表中包含大量字段
 - 启用了二级缓存, 且二级缓存中可能已经包含了待查询的对象

管理 Session

- Hibernate 自身提供了三种管理 Session 对象的方法
 - **Session 对象的生命周期与本地线程绑定**
 - Session 对象的生命周期与 JTA 事务绑定
 - Hibernate 委托程序管理 Session 对象的生命周期
- 在 Hibernate 的配置文件中,
hibernate.current_session_context_class 属性用于指定 Session 管理方式, 可选值包括
 - **thread**: Session 对象的生命周期与本地线程绑定
 - jta*: Session 对象的生命周期与 JTA 事务绑定
 - managed: Hibernate 委托程序来管理 Session 对象的生命周期

Session 对象的生命周期与本地线程绑定

- 如果把 Hibernate 配置文件的 `hibernate.current_session_context_class` 属性值设为 `thread`, Hibernate 就会按照与本地线程绑定的方式来管理 Session
- Hibernate 按一下规则把 Session 与本地线程绑定
 - 当一个线程(threadA)第一次调用 SessionFactory 对象的 `getCurrentSession()` 方法时, 该方法会创建一个新的 Session(sessionA) 对象, 把该对象与 threadA 绑定, 并将 sessionA 返回
 - 当 threadA 再次调用 SessionFactory 对象的 `getCurrentSession()` 方法时, 该方法将返回 sessionA 对象
 - 当 threadA 提交 sessionA 对象关联的事务时, Hibernate 会自动flush sessionA 对象的缓存, 然后提交事务, 关闭 sessionA 对象. 当 threadA 撤销 sessionA 对象关联的事务时, 也会自动关闭 sessionA 对象
 - 若 threadA 再次调用 SessionFactory 对象的 `getCurrentSession()` 方法时, 该方法会又创建一个新的 Session(sessionB) 对象, 把该对象与 threadA 绑定, 并将 sessionB 返回

批量处理数据

- 批量处理数据是指在一个事务中处理大量数据.
- 在应用层进行批量操作, 主要有以下方式:
 - 通过 Session
 - 通过 HQL
 - 通过 StatelessSession
 - 通过 **JDBC API**

通过 Session 来进行批量操作

- Session 的 save() 及 update() 方法都会把处理的对象存放在自己的缓存中. 如果通过一个 Session 对象来处理大量持久化对象, 应该**及时从缓存中清空已经处理完毕并且不会再访问的对象**. 具体的做法是**在处理完一个对象或小批量对象后, 立即调用 flush() 方法刷新缓存, 然后在调用 clear() 方法清空缓存**
- 通过 Session 来进行处理操作会受到以下约束
 - 需要在 Hibernate 配置文件中设置 JDBC 单次批量处理的数目, 应保证每次向数据库发送的批量的 SQL 语句数目与 batch_size 属性一致
 - 若对象采用 “identity” 标识符生成器, 则 Hibernate 无法在 JDBC 层进行批量插入操作
 - 进行批量操作时, 建议关闭 Hibernate 的二级缓存

通过 Session 来进行批量操作

- 批量插入数据:

```
News news = null;
for(int i = 0; i < 100000; i++){
    news = new News();
    news.setTitle("--" + i);

    session.save(news);
    if((i + 1) % 20 == 0){
        session.flush();
        session.clear();
    }
}
```

通过 Session 来进行批量操作

- 批量更新: 在进行批量更新时, 如果一下子把所有对象都加载到 Session 缓存, 然后再缓存中一一更新, 显然是不可取的
- 使用可滚动的结果集 `org.hibernate.ScrollableResults`, **该对象中实际上并不包含任何对象, 只包含用于在线定位记录的游标**. 只有当程序遍历访问 `ScrollableResults` 对象的特定元素时, 它才会到数据库中加载相应的对象.
- `org.hibernate.ScrollableResults` 对象由 Query 的 `scroll` 方法返回

```
ScrollableResults sr = session.createQuery("FROM News").scroll();
```

```
int count = 0;
while(sr.next()){
    News n = (News) sr.get(0);
    n.setTitle(n.getTitle() + "*****");

    if(((count++) + 1) % 100 == 0){
        session.flush();
        session.clear();
    }
}
```

通过 HQL 来进行批量操作

- 注意: HQL 只支持 INSERT INTO ... SELECT 形式的插入语句, 但不支持 INSERT INTO ... VALUES 形式的插入语句. 所以使用 HQL 不能进行批量插入操作.

通过StatelessSession来进行批量操作

- 从形式上看，StatelessSession与session的用法类似。
StatelessSession与session相比，有以下区别：
 - StatelessSession没有缓存，通过StatelessSession来加载、保存或更新后的对象处于游离状态。
 - StatelessSession不会与Hibernate的第二级缓存交互。
 - 当调用StatelessSession的save()、update()或delete()方法时，这些方法会立即执行相应的SQL语句，而不会仅计划执行一条SQL语句
 - StatelessSession不会进行脏检查，因此修改了Customer对象属性后，还需要调用StatelessSession的update()方法来更新数据库中数据。
 - StatelessSession不会对关联的对象进行任何级联操作。
 - 通过同一个StatelessSession对象两次加载OID为1的Customer对象，得到的两个对象内存地址不同。
 - StatelessSession所做的操作可以被Interceptor拦截器捕获到，但是会被Hibernate的事件处理系统忽略掉。

