J2EE与中间件技术

——Java Persistence API

会话Bean

- 应用逻辑组件——代表为一个用户而执行的操作；不能直接表示持久数据；无法提供持久数据的某些共享特征；当服务器重启或出现系统异常，不能继续存在
  - 例如：购物车、信用卡验证

持久的概念

- Java对象序列化：将对象序列化，应用于网络通信或简单的持久存储
- 对象/关系型数据库映射：使用关系型数据库持久存储Java对象（利用JDBC ）
- 对象数据库：完整的对象被持久存储

Java Persistence API

- The Java Persistence API provides an object/relational mapping facility to Java developers for managing relational data in Java applications.

Entities

- An entity is a lightweight persistence domain object.
- Typically an entity represents a table in a relational database, and each entity instance corresponds to a row in that table.

对象/关系型数据库映射

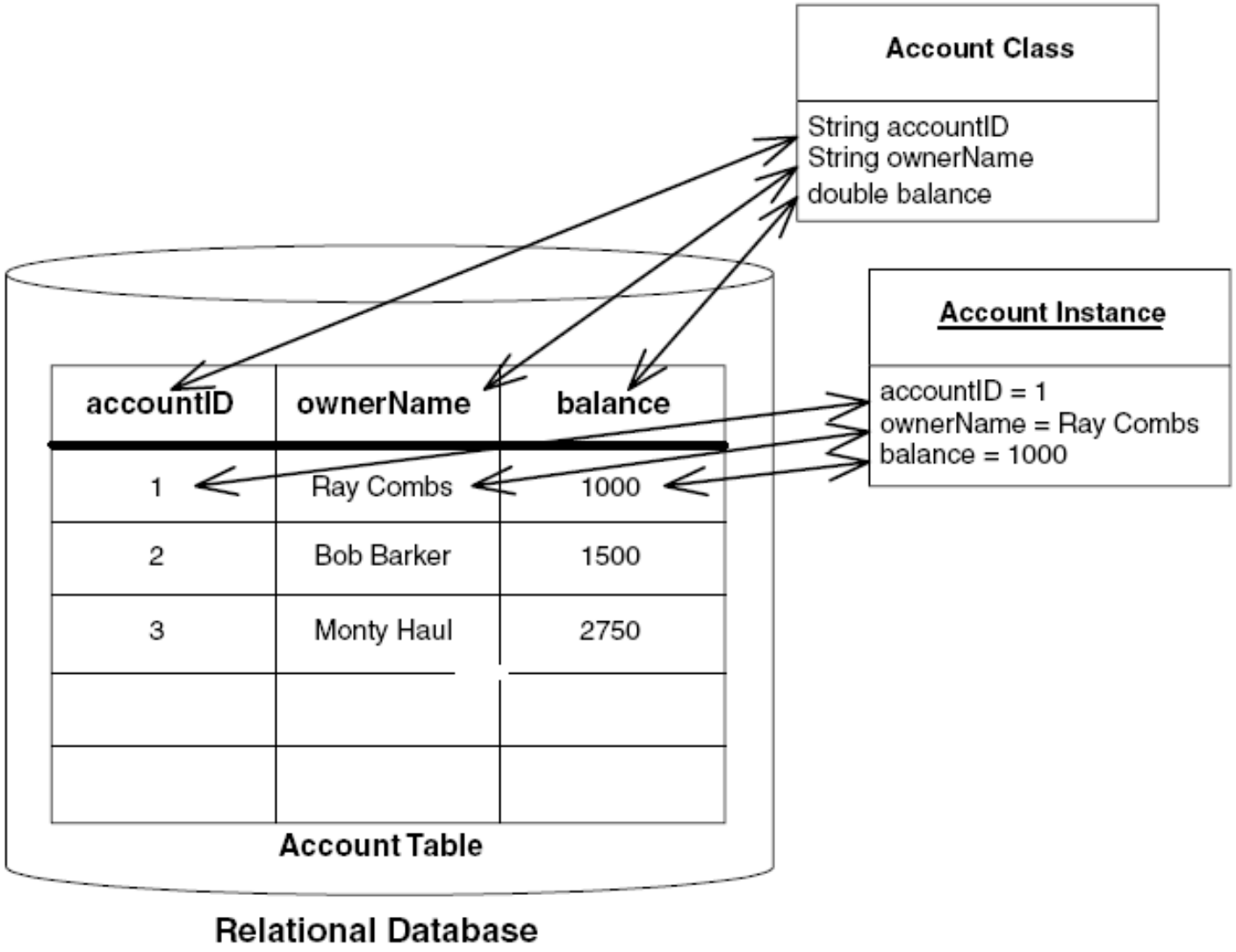


Figure 5.2 An example of object-relational mapping.

Entities

- 持久数据组件——内存中的对象，对应到数据库中的一个视图；一种持久性的、事务性的以及可以共享的组件，多个客户机可以同时使用其中的业务数据
  - 例如：顾客、订单、产品、信用卡

持久数据组件

- 为什么要把商务数据当作对象来处理，而不是处理数据库中的原始数据？
  - 1. 把数据视为对象，能够方便的操作和管理它们；
  - 2. 可以把相关的数据聚合在一起成为一个统一的对象；
  - 3. 可以把一些简单的方法和那些数据联系在一起；
  - 4. 可以将这些数据存储到缓存中，提高性能
  - 5. 可以从一个应用服务器上得到隐含的中间件服务：如关联、事务、网络访问能力和安全服务

Entity Classes

- The primary programming artifact of an entity is the entity class, although entities can use helper classes.
- The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

Entity Classes

- The class must be annotated with the javax.persistence.Entity annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must not be declared final. No methods or persistent instance variables must be declared final.
- If an entity instance be passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the Serializable interface.
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- Persistent instance variables must be declared private, protected, or package-private, and can only be accessed directly by the entity class's methods. Clients must access the entity's state through accessor or business methods.

Persistent Fields and Properties

- The persistent state of an entity can be accessed either through the entity's instance variables or through JavaBeans-style properties.
- Entities may either use persistent fields or persistent properties.
  - If the mapping annotations are applied to the entity's instance variables, the entity uses persistent fields.
  - If the mapping annotations are applied to the entity's getter methods for JavaBeans-style properties, the entity uses persistent properties.

Persistent Fields

- If the entity class uses persistent fields, the Persistence runtime accesses entity class instance variables directly.
- All fields not annotated javax.persistence.Transient or not marked as Java transient will be persisted to the data store.
- The object/relational mapping annotations must be applied to the instance variables.

Persistent Properties

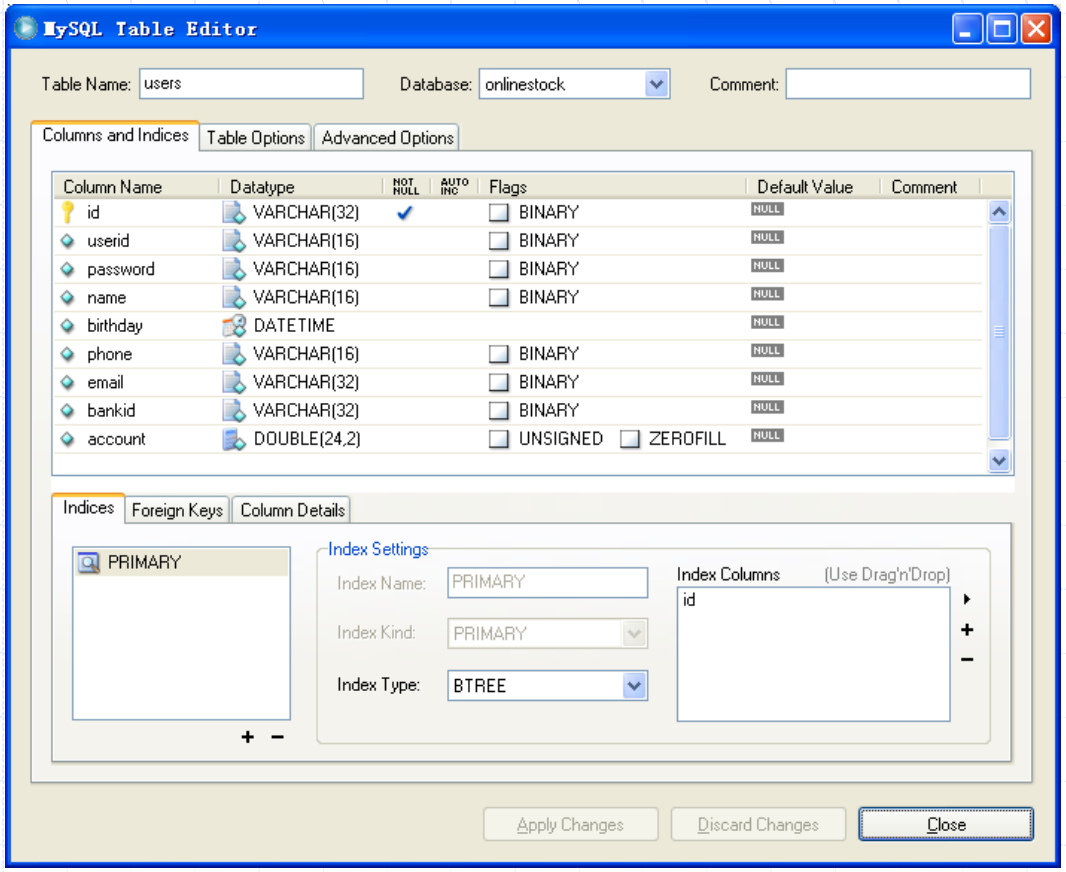
- If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components.
- JavaBeans-style properties use getter and setter methods that are typically named after the entity class's instance variable names.
- If the property is a boolean, you may use is*Property* instead of get*Property*.
- The object/relational mapping annotations for must be applied to the getter methods.

Primary Keys in Entities

- Each entity has a unique object identifier.
- The unique identifier, or *primary key*, enables clients to locate a particular entity instance. Every entity must have a primary key.
- Simple primary keys use the javax.persistence.Id annotation to denote the primary key property or field.
- Composite primary keys must be defined in a primary key class. Composite primary keys are denoted using the javax.persistence.EmbeddedId and javax.persistence.IdClass annotations.

示例

- 数据库onlinestock
- 表users


MySQL Table Editor
Table Name: users
Database: onlinestock
Comment:
Columns and Indices
Table Options
Advanced Options
Column Name
Datatype
NOT NULL
AUTO INC
Flags
Default Value
Comment
id VARCHAR(32) BINARY NULL
userid VARCHAR(16) BINARY NULL
password VARCHAR(16) BINARY NULL
name VARCHAR(16) BINARY NULL
birthday DATETIME NULL
phone VARCHAR(16) BINARY NULL
email VARCHAR(32) BINARY NULL
bankid VARCHAR(32) BINARY NULL
account DOUBLE(24,2) UNSIGNED ZEROFILL NULL
Indices
Foreign Keys
Column Details
PRIMARY
Index Settings
Index Name: PRIMARY
Index Kind: PRIMARY
Index Type: BTREE
Index Columns (Use Drag'n'Drop)
id
Apply Changes
Discard Changes
Close

示例

- User：与users表进行映射的实体Bean类

User

```
……
@Entity
@Table(name="users")
public class User implements Serializable {
    @Id
    private String id;
    private double account;
    private String bankid;
    private Date birthday;
    private String email;
    private String name;
    private String password;
    private String phone;
    private String userid;
    public User() {
    }
```

```
    public String getId() {
        return this.id;
    }

    public void setId(String id) {
        this.id = id;
    }
    ……
}
```

- 不为持久字段和属性指定@Column注释，将假定到同名字段和属性的数据库列的默认映射

实体

- 一个实体的实例是一个对应到数据库中的视图：
  - 更新内存中的实体实例，数据库也自动被更新——java对象与数据库的同步是由容器自动完成的
- 一个实体的几个实例可能代表同一底层数据：
  - 例如：许多不同的客户端浏览器同时访问一个产品目录——保证每一个Bean实例的数据都是最新的

Managing Entities

- Entities are managed by the entity manager.
- The entity manager is represented by javax.persistence.EntityManager instances.
- Each EntityManager instance is associated with a persistence context.

The Persistence Context

- A persistence context is **a set of managed entity instances** that exist in a particular data store.
- The **EntityManager interface** defines the methods that are used to interact with the persistence context.

The EntityManager Interface

- The EntityManager API creates and removes persistent entity instances, finds entities by the entity's primary key, and allows queries to be run on entities.

Container-Managed EntityManagers

- With a *container-managed entity manager*, an EntityManager instance's persistence context is automatically propagated by the container to all application components that use the EntityManager instance within a single Java Transaction Architecture (JTA) transaction.
- To complete a JTA transaction, these components usually need access to a single persistence context. This occurs when an EntityManager is injected into the application components by means of the javax.persistence.PersistenceContext annotation.

```
@PersistenceContext
EntityManager em;
```

Application-Managed EntityManagers

- With *application-managed entity managers*, on the other hand, the persistence context is not propagated to application components, and the life cycle of EntityManager instances is managed by the application.
- Applications create EntityManager instances in this case by using the createEntityManager method of javax.persistence.EntityManagerFactory.
- To obtain an EntityManager instance, you first must obtain an EntityManagerFactory instance by injecting it into the application component by means of the javax.persistence.PersistenceUnit annotation.

```
@PersistenceUnit
EntityManagerFactory emf;
```

Finding Entities Using the EntityManager

- The EntityManager.find method is used to look up entities in the data store by the entity's primary key.

```
@PersistenceContext
EntityManager em;
public void enterOrder(int custID, Order
newOrder) {
    Customer cust = em.find(Customer.class, custID);
    ......
}
```

Managing an Entity Instance's Life Cycle

- You manage entity instances by invoking operations on the entity by means of an EntityManager instance.
- Entity instances are in one of four states: new, managed, detached, or removed.
  - **New** entity instances have no persistent identity and are not yet associated with a persistence context.
  - **Managed** entity instances have a persistent identity and are associated with a persistence context.
  - **Detached** entity instances have a persistent identity and are not currently associated with a persistence context.
  - **Removed** entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

Entity Life Cycle

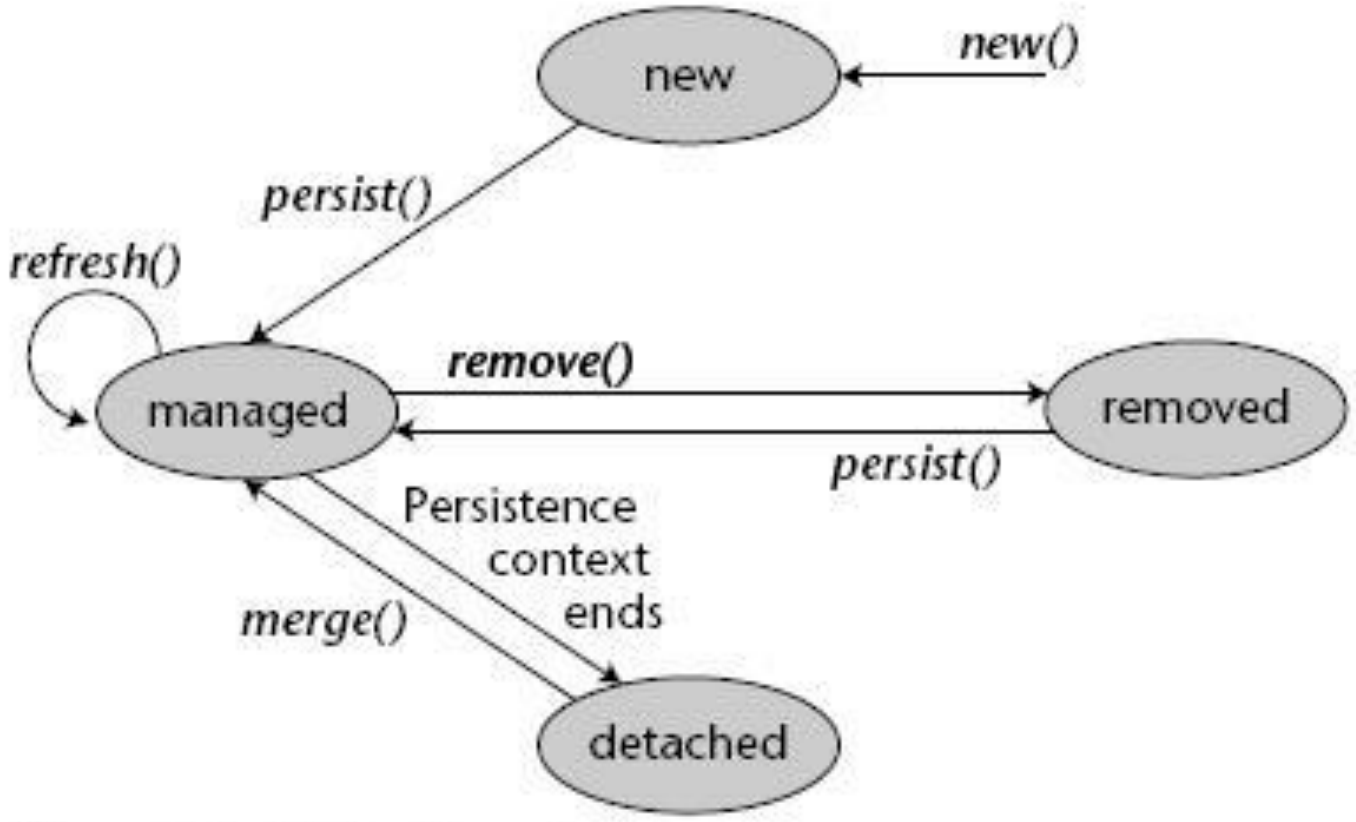


Figure 6.3 Entity life cycle.

new

- 通过new生成一个实体对象如：

  User user=new User(“001”,“xyz”,……);

- user通过JVM获得了一块内存空间，但是并没有保存进数据库，还没有纳入JPA EntityManager管理中
- 在数据库中不存在一条与它对应的记录

managed

- 纳入JPA EntityManager管理中的对象
- new状态，可通过persist()方法把user与数据库相关联，成为**持久化对象**
- 或使用find()方法，得到持久化对象
- 在数据库中存在一条与它对应的记录，并拥有一个持久化标识（identifier）
- 对持久化对象的操作，**影响数据库**

detached

- 游离对象
  - 例如：find()方法调用后，可关闭EntityManager，成为游离对象
    - 如：em.clear();
  - 对游离对象的操作，不影响数据库
  - 和new状态的区别
    - 在数据库中可能还存在一条与它对应的记录，只是现在这个游离对象脱离了JPA EntityManager的管理
- 游离对象转为持久对象
  - 调用merge()方法

removed

- remove()方法
- 删除数据库中的记录
- 在适当的时候被垃圾回收

Managing an Entity Instance's Life Cycle

- Persisting Entity Instances
  - New entity instances become managed and persistent by invoking the persist method.
  - or by a cascading persist operation invoked from related entities that have the cascade=PERSIST or cascade=ALL elements set in the relationship annotation.
  - This means the entity's data is stored to the database when the transaction associated with the persist operation is completed.
    - If the entity is already managed, the persist operation is ignored, although the persist operation will cascade to related entities that have the cascade element set to PERSIST or ALL in the relationship annotation.
    - If persist is called on a removed entity instance, it becomes managed.
    - If the entity is detached, persist will throw an IllegalArgumentException, or the transaction commit will fail.

Managing an Entity Instance's Life Cycle

- Removing Entity Instances
  - Managed entity instances are removed by invoking the remove method.
  - or by a cascading remove operation invoked from related entities that have the cascade=REMOVE or cascade=ALL elements set in the relationship annotation.
  - The entity's data will be removed from the data store when the transaction is completed, or as a result of the flush operation.
    - If the remove method is invoked on a new entity, the remove operation is ignored, although remove will cascade to related entities that have the cascade element set to REMOVE or ALL in the relationship annotation.
    - If remove is invoked on a detached entity it will throw an IllegalArgumentException, or the transaction commit will fail.
    - If remove is invoked on an already removed entity, it will be ignored.

Managing an Entity Instance's Life Cycle

- Creating Queries
  - The EntityManager.createQuery and EntityManager.createNamedQuery methods are used to query the datastore using Java Persistence query language queries.
    - The createQuery method is used to create *dynamic queries*, queries that are defined directly within an application's business logic.
    - The createNamedQuery method is used to create *static queries*, queries that are defined in metadata using the javax.persistence.NamedQuery annotation.

Persistence Units

- A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application.
- Persistence units are defined by the persistence.xml configuration file.
- Each persistence unit must be identified with a name that is unique to the persistence unit's scope.
- If you package the persistent unit as a set of classes in an EJB JAR file, persistence.xml should be put in the EJB JAR's META-INF directory.

persistence.xml

- persistence.xml defines one or more persistence units.
- Example:

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description>
            This unit manages orders and customers.
        </description>
        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
        <jar-file>MyOrderApp.jar</jar-file>
        <class>com.widgets.Order</class>
        <class>com.widgets.Customer</class>
    </persistence-unit>
</persistence>
```

示例

- persistence.xml：实体Bean的配置文件

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
version="1.0">
    <persistence-unit name="nju">
        <jta-data-source>java:/MySqlDS</jta-data-source>
        <properties>
            <property name="hibernate.dialect"
           value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
        </properties>
    </persistence-unit>
 </persistence>
```

JBoss数据源配置

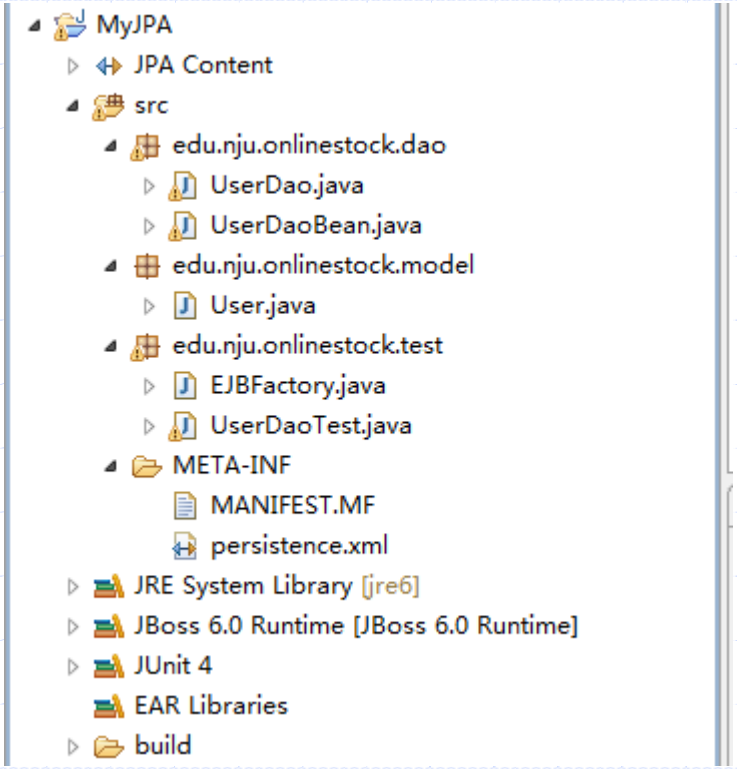
- Lab-Jboss-2.ppt(JBoss7)

Accessing Entities

- Since an entity cannot be accessed remotely, the only option that we have is to deploy it locally and use it from either J2SE code outside a container, or from session or message-driven beans living in EJB container.

示例

- UserDAORemote：使用实体Bean的会话Bean的业务接口
- UserDAO：使用实体Bean的会话Bean的实现类
- UserDAOTest：使用JUnit创建的EJB测试类


MyJPA
JPA Content
src
edu.nju.onlinestock.dao
UserDao.java
UserDaoBean.java
edu.nju.onlinestock.model
User.java
edu.nju.onlinestock.test
EJBFactory.java
UserDaoTest.java
META-INF
MANIFEST.MF
persistence.xml
JRE System Library [jre6]
JBoss 6.0 Runtime [JBoss 6.0 Runtime]
JUnit 4
EAR Libraries
build

UserDAORemote

```
……
@Remote
public interface UserDAORemote {
    public boolean insertUser(String id,String userID, String
    password, String name, Date birthday, String phone, String
    email,String bankID,double account);
    public String getUserNameByID(String userID);
    public boolean updateUser(User user);
    public User getUserByID(String id);
    @SuppressWarnings("unchecked")
    public List getUserList();
}
```

UserDAO

```
.....

@Stateless
public class UserDAO implements UserDAORemote {
    @PersistenceContext
    protected EntityManager em;

    public User getUserByID(String id) {
        User user = em.find(User.class, id);
        return user;
    }
    public String getUserNameByID(String id) {
        User user = em.find(User.class, id);
        return user.getUserName();
    }
```

```
public List getUserList() {
    try{
        Query query = em.createQuery("from User u order by userid
        asc");
        List list =query.getResultList();
        em.clear();//在处理大量实体的时候，如果不把已经处理过的实体
        从EntityManager中分离出来，将会消耗大量的内存；此方法分离内
        存中受管理的实体Bean，让VM进行垃圾回收
        return list;
    }catch(Exception e){
        e.printStackTrace();
        return null;
    }
}
```

```java
public boolean insertUser(String id, String userID,
String password, String name, Date birthday, String phone,
String email, String bankID, double account) {
    try{
        User user = new User(); //new状态，在数据库中不存在一条与它对应的记录
        user.setId(id);
        user.setUserid(userID);
        user.setPassword(password);
        user.setName(name);
        user.setBirthday(birthday);
        user.setPhone(phone);
        user.setEmail(email);
        user.setBankid(bankID);
        user.setAccount(account);
        em.persist(user); //managed 状态，保存Entity到数据库中
    }catch(Exception e){
        e.printStackTrace();
        return false;
    }
    return true;
}
```

```
    public boolean updateUser(User user) {
        try{
            em.merge(user); //容器决定flush时，数据将同步到数据库中
        }catch(Exception e){
            e.printStackTrace();
            return false;
        }
        return true;

    }

}
```

UserDAOTest(以Jboss6为例)

```
……
public class UserDAOTest {
    private static UserDAORemote dao;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        InitialContext ctx = null;
        try {
            Properties props = new Properties();
            props.setProperty("java.naming.factory.initial",
            "org.jnp.interfaces.NamingContextFactory");
            props.setProperty("java.naming.provider.url", "localhost:1099");
            props.setProperty("java.naming.factory.url.pkgs",
            "org.jboss.naming:org.jnp.interfaces");
            ctx = new InitialContext(props);
        } catch (NamingException e) {
            e.printStackTrace();
        }
        dao=(UserDAORemote) ctx.lookup("/UserDAO/remote");
    }
```

```
@Test
public void testInsertUser(){
    assertTrue(dao.insertUser("1234567890","user001", "123", "abcd", new Date(1980-
    01-01),"123456","user1@nju.edu.cn", "123123", 100.0));
}

@Test
public void testUpdateUser(){
    User user=dao.getUserByID( “1234567890");
    user.setUserName( “def");
    assertNotNull(user);
    assertTrue(dao.updateUser(user));
}
```

```
    @Test
    public void testGetUserByID(){
        assertNotNull(dao.getUserByID( “1234567890"));
    }

    @SuppressWarnings("unchecked")
    @Test
    public void testGetUserList(){
        List list=dao.getUserList();
        assertNotNull(list);
        assertEquals(false,list.isEmpty());
    }
}
```

银行账户实体Bean

- 实体Bean类：Account
- persistence.xml：实体Bean的配置文件（与User相同）
- AccountDAORemote：使用实体Bean的会话Bean的业务接口
- AccountDAO：使用实体Bean的会话Bean的实现类
- AccountDAOClient：客户端应用程序

ACCOUNTtbl表

- create table accounttbl (accountid varchar(50) primary key, name varchar(50), balance double precision);

Account

```
……
@Entity
@Table(name="accounttbl")
public class Account implements Serializable {
    private String accountid;
    private String name;
    private double balance;
```

```
    @Id
    public String getAccountid(){
        return accountid;
    }
    @Column(nullable=false, length=50)
    public String getName() {
        return name;
    }
    @Column(nullable=false)
    public double getBalance() {
        return balance;
    }

    public void setAccountid(String accountid){
        this.accountid=accountid;
    }
    public void setName(String Name){
        this.name=Name;
    }
    public void setBalance(double balance){
        this.balance=balance;
    }
}
```

AccountDAORemote

```
……
@Remote
public interface AccountDAORemote {
    public Account getAccountByAccountid(String accountid);
    @SuppressWarnings("unchecked")
    public List getAccountByCustomName(String customName);
    public double getTotalBankValue();
    public void debit(Account account, double amount) throws
    BalanceException;
    public void credit(Account account, double amount);
    public boolean insertAccount(String accountid, String name,
    double balance);
}
```

AccountDAO

```
……

@Stateless
public class AccountDAO implements AccountDAORemote {
    @PersistenceContext
    protected EntityManager em;
```

```
public boolean insertAccount(String accountid,
String name, double balance) {
    try{
        Account account = new Account();
        account.setAccountid(accountid);
        account.setName(name);
        account.setBalance(balance);
        em.persist(account);
    }catch(Exception e){
        e.printStackTrace();
        return false;
    }
    return true;
}
```

```
public void credit(Account account, double amount) {
    double balance= account.getBalance();
    balance += amount;
    account.setBalance(balance);
    em.merge(account);
}

public void debit(Account account, double amount) throws
BalanceException {
    double balance= account.getBalance();
    if (balance< amount)
        throw new BalanceException();
    balance -= amount;
    account.setBalance(balance);
    em.merge(account);
}
```

```
private List list=null;
public List getAccountByCustomName(String name) {
    try{
        Query query = em.createQuery("from Account a where a.name=name");
        list =query.getResultList();
        em.clear();
        return list;
    }catch(Exception e){
        e.printStackTrace();
        return null;
    }
}

public Account getAccountByAccountid(String accountid) {
    Account account = em.find(Account.class, accountid);
    return account;
}
```

```
public double getTotalBankValue() {
    try{
        Query query =em.createQuery("select sum(a.balance)
        from Account a");
        Double total=(Double) query.getSingleResult();
        em.clear();
        return total;
    }catch(Exception e){
        e.printStackTrace();
        return 0;
    }
}
```

BalanceException

```
……

public class BalanceException extends Exception {
    public BalanceException() { }
    public BalanceException(String msg) {
        super(msg);
    }
    public BalanceException(Exception e) {
        super(e.toString());
    }
}
```

AccountDAOClient（以JBoss6为例）

……

```
public class AccountDAOClient {
    private static AccountDAORemote dao;
    public static void main(String[] args) throws
    BalanceException  {
```

```
dao= (AccountDAORemote)EJBFactory.getEJB("AccountDAO/remote");

System.err.println("Total of all accounts in bank
initially="+dao.getTotalBankValue());

dao.insertAccount("123-456-7890","John Smith",100);

Account account=dao.getAccountByAccountid("123-456-7890");
System.err.println("Initial Balance="+account.getBalance());
dao.credit(account,100);

account=dao.getAccountByAccountid("123-456-7890");
System.err.println("After crediting 100,account
Balance="+account.getBalance());
System.err.println("Total of all accounts in bank
now="+dao.getTotalBankValue());
```

```
        List list=dao.getAccountByCustomName("John Smith");
        if (list.isEmpty())
            System.err.println("Could not find John Smith.");
        else{
            Iterator iterator=(Iterator) list.iterator();
            while (iterator.hasNext()){
                account=(Account) iterator.next();
                try{
                    dao.debit(account,250);
                }catch(BalanceException be){
                    System.err.println("Now Trying to withdraw $250, which is more
                than currently available. This should generate an exception.");
                }
                System.err.println("After debiting 250,account
                Balance="+account.getBalance());
                System.err.println("Total of all accounts in bank
                now="+dao.getTotalBankValue());
            }
        }
    }
}
```

运行结果

```
Total of all accounts in bank initially=0.0
Initial Balance=100.0
After crediting 100,account Balance=200.0
Total of all accounts in bank now=200.0
Now Trying to withdraw $250, which is more than currently
available. This should generate an exception.
After debiting 250,account Balance=200.0
Total of all accounts in bank now=200.0
```

Remote接口

- 客户端采用Remote接口来调用Enterprise Bean：
  - Stub、Skeleton、网络、参数整理/再整理
  - 生成Bean是非常慢的、效率不高

Local 接口

- 快速、高效
- 客户端是运行在同一个EJB容器中的：采用Local接口来调用Enterprise Bean
  - 没有Stub、Skeleton代理，以更快的方式生成Bean

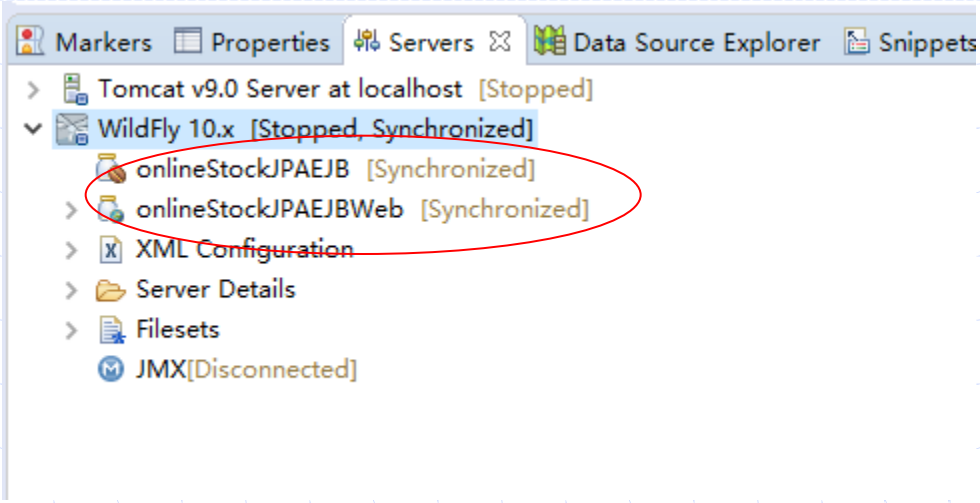
```
……
@Local
public interface UserDao {
……
}
```

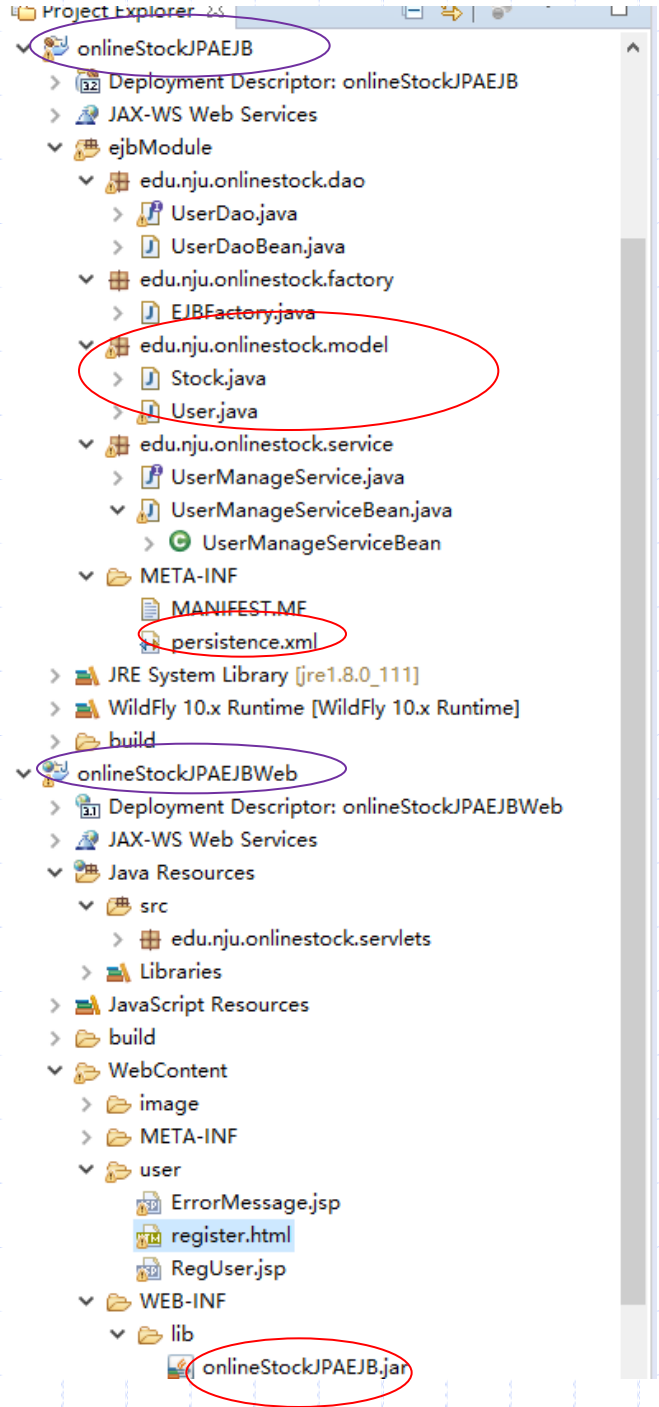
User2Servlet.java

```
……
@WebServlet("/User2Servlet ")
public class User2Servlet extends
HttpServlet {
    @EJB
    Private UserDao dao;
    //以一种“注入”的方式获得一个bean的业务
    接口引用
    ……
```

```
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws
    ServletException, IOException {
        dao.insertUser(1234567892,"user001", "123", "abcd", new
        Date(1980-01-01),"123456","user1@nju.edu.cn", "123123",
        100.0);
        User user=dao.getUserByID(1234567891);
        user.setName("def");
        dao.updateUser(user);
    }
    ......
}
```

onlineStock项目（wildfly10）





以注册用户为例

- Web Project
- EJB Project
  - ……
  - edu.nju.onlinestock.dao
    - UserDaoBean: Session Beans
      - JPA EntityManager
  - edu.nju.onlinestock.model
    - User
      - Entity Beans

作业6

- 修改作业5中数据访问层和Model的设计
  - edu.nju.onlinestock.model
    - Entity Beans
  - edu.nju.onlinestock.dao
    - Session Beans
      - JPA EntityManager

The Java Persistence Query Language

- The Java Persistence query language defines queries for entities and their persistent state.
- 语法类似于SQL，面向对象，用于查询实体Bean的查询语言

Select Statements

- A select query has six clauses: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY.
- The SELECT and FROM clauses are required.
- QL_statement ::= select_clause from_clause [where_clause][groupby_clause][having_clause][orderby_clause]

SELECT and FROM clauses

- The SELECT clause defines the types of the objects or values returned by the query.
- The FROM clause defines the scope of the query by declaring one or more identification variables, which can be referenced in the SELECT and WHERE clauses.
  - An identification variable represents one of the following elements:
    - The abstract schema name of an entity
    - An element of a collection relationship
    - An element of a single-valued relationship
    - A member of a collection that is the multiple side of a one-to-many relationship

A Basic Select Query

- SELECT p FROM Player p
  - Data retrieved: All players.
  - The FROM clause declares an identification variable named p.
  - The Player element is the abstract schema name of the Player entity.

Aggregate Functions in the SELECT Clause

- The result of a query may be the result of an aggregate function.
- AVG, COUNT, MAX, MIN, SUM
- Example:
  - SELECT AVG(o.quantity) FROM Order o
  - SELECT COUNT(o) FROM Order o
  - select sum(a.balance) from Account a

WHERE clause

- The WHERE clause is a conditional expression that restricts the objects or values retrieved by the query.
- Example:
  - from Account a where a.name=name

Eliminating Duplicate Values

- SELECT DISTINCT p FROM Player p WHERE p.position = ?1
  - Data retrieved: The players with the position specified by the query's parameter.
  - The DISTINCT keyword eliminates duplicate values.
  - The WHERE clause restricts the players retrieved by checking their position, a persistent field of the Player entity.
  - The ?1 element denotes the input parameter of the query.

Using Named Parameters

- SELECT DISTINCT p FROM Player p WHERE p.position = :position AND p.name = :name
  - Data retrieved: The players having the specified positions and names.
  - The position and name elements are persistent fields of the Player entity.
  - The WHERE clause compares the values of these fields with the named parameters of the query, set using the Query.setNamedParameter method.

GROUP BY and HAVING clause

- The GROUP BY clause groups query results according to a set of properties.
- The HAVING clause is used with the GROUP BY clause to further restrict the query results according to a conditional expression.

ORDER BY clause

- The ORDER BY clause sorts the objects or values returned by the query into a specified order.
- The ASC keyword specifies ascending order (the default), and the DESC keyword indicates descending order.
- When using the ORDER BY clause, the SELECT clause must return an orderable set of objects or values.
- Example:
  - from User u order by userid asc

数据库表关系

- Entity——映射数据库表的Java对象，必须能够体现出数据库表之间的关系
  - 从一个实体到另一个实体的引用
  - 保证数据的完整性，数据的增加、修改和删除必须符合规定的数据约束

关系

- 基数性
- 方向性
- 聚合—组合和级联删除
- 惰性载入
- 引用的完整性

Multiplicity in Entity Relationships

- There are four types of multiplicities: one-to-one, one-to-many, many-to-one, and many-to-many.

1:1关系

- 例如：订单与出货单

OrderPK	OrderName	Shipment ForeignPK
12345	Software Order	10101

ShipmentPK	City	ZipCode
10101	Austin	78727

Figure 11.2 A possible one-to-one cardinality database schema.

- **Foreign Key**：数据库表中的一个列，这个列是另一个数据库表的主键
- **EJB**容器使用外键实现数据库的关系

One-to-one

- Each entity instance is related to a single instance of another entity.
- One-to-one relationships use the javax.persistence.OneToOne annotation on the corresponding persistent property or field.

1:N关系

- 例如：公司与雇员

CompanyPK	Name	Employee FKs
12345	The Middleware Company	<Vector BLOB>

EmployeePK	Name	Sex
20202	Ed	M
20203	Floyd	M

Figure 11.3 A possible one-to-many cardinality database schema.

1:N关系

- 另一种模式

CompanyPK	Name
12345	The Middleware Company

EmployeePK	Name	Sex	Company
20202	Ed	M	12345
20203	Floyd	M	12345

Figure 11.4 Another one-to-many cardinality database schema.

One-to-many

- An entity instance can be related to multiple instances of the other entities.
- One-to-many relationships use the javax.persistence.OneToMany annotation on the corresponding persistent property or field.

Many-to-one

- Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship.
- Many-to-one relationships use the javax.persistence.ManyToOne annotation on the corresponding persistent property or field.

M:N关系

- 例如：学生与课程，使用一个中间的关系表

StudentPK	StudentName
10101	Joe Student

EnrollmentPK	StudentPK	CoursePK
12345	10101	20202

CoursePK	CourseName
20202	EJB for Architects

Figure 11.5 A possible many-to-many cardinality database schema.

Many-to-many

- The entity instances can be related to multiple instances of each other.
- Many-to-many relationships use the javax.persistence.ManyToMany annotation on the corresponding persistent property or field.

Direction in Entity Relationships

- The direction of a relationship can be either bidirectional or unidirectional.
- A bidirectional relationship has both an owning side （维护端/主控方）and an inverse side（被维护端/被控方）.
- A unidirectional relationship has only an owning side.
- The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

Bidirectional Relationships

- Each entity has a relationship field or property that refers to the other entity.
- The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation.
- The many side of many-to-one bidirectional relationships must not define the mappedBy element.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many bidirectional relationships either side may be the owning side.

Unidirectional Relationships

- In a *unidirectional* relationship, only one entity has a relationship field or property that refers to the other.
- 不需要定义mappedBy属性.

Queries and Relationship Direction

- Java Persistence query language queries often navigate across relationships.
- The direction of a relationship determines whether a query can navigate from one entity to another.

Queries That Navigate to Related Entities

- Data retrieved: All players who belong to a team.
  - SELECT DISTINCT p FROM Player p, IN(p.teams) t
  - SELECT DISTINCT p FROM Player p JOIN p.teams t
  - SELECT DISTINCT p FROM Player p WHERE p.team IS NOT EMPTY

Joins

- A **LEFT JOIN** or LEFT OUTER JOIN retrieves a set of entities where matching values in the join condition may be absent.
- A **FETCH JOIN** is a join operation that returns associated entities as a side-effect of running the query.
- Example:
  - SELECT d FROM Department d LEFT JOIN FETCH d.employees WHERE d.deptno = 1
  - The query returns a set of departments, and as a side-effect, the associated employees of the departments

Cascade Deletes and Relationships

- Entities that use relationships often have dependencies on the existence of the other entity in the relationship.
- Cascade delete relationships are specified using the cascade=REMOVE element specification for @OneToOne and @OneToMany relationships.

1：N关系示例

- 一个用户对应多个文件
- 一：inverse side,关系被维护端
  - 使用OneToMany
  - 需要定义mappedBy属性
- 多：owning side,关系维护端
  - 使用ManyToOne
  - 外键JoinColumn信息在这个类里定义

User表

- CREATE TABLE `Usertbl` (`UserID` varchar(50) NOT NULL, `UserName` varchar(10) NOT NULL, `UserMail` varchar(50) NOT NULL, `UserPassword` varchar(50) NOT NULL, `UserType` int default 0, PRIMARY KEY (`UserID`));

File表

- CREATE TABLE `Filetbl` (`FileID` BIGINT NOT NULL, `FileName` varchar(255) NOT NULL, `FilePath` varchar(255) NOT NULL, `FileType` varchar(10), `FileOwner` varchar(50) NOT NULL , `FileSubject` varchar(100) NOT NULL, PRIMARY KEY (`FileID`), FOREIGN KEY(`FileOwner`) REFERENCES Usertbl(`UserID`));

- Insert into `Usertbl` values(`user1`, `测试用户1`, `user1@nju.edu.cn`, `password`, 0);
- Insert into `Usertbl` values(`user2`, `测试用户2`, `user2@nju.edu.cn`, `password`, 0);

- Insert into `Filetbl` values(1, `课程表.doc`, `d:\\files`, `word`, `user1`, `教学`);
- Insert into `Filetbl` values(2, `基金项目指南.doc`, `d:\\files`, `word`, `user1`, `项目`);

- User类：实体Bean
- File类：实体Bean
- 且形成双向关系
- 二者有级联关系
  - 删除User将同时删除对应的File

User

```
……
@Entity
@Table(name="Usertbl")
public class User implements Serializable {
    private String userID;
    private String userName;
    private String userMail;
    private String userPassword;
    private int userType=0;
    private Set<File> files=new HashSet<File>();

    @Id
    public String getUserID(){
        return userID;
    }
    public void setUserID(String userID){
        this.userID=userID;
    }
```

```
@Column(nullable=false, length=50)
public String getUserMail(){
        return userMail;
}
public void setUserMail(String userMail){
        this.userMail=userMail;
}
@Column(nullable=false, length=50)
public String getUserName(){
        return userName;
}
public void setUserName(String userName){
        this.userName=userName;
}
@Column(nullable=false, length=50)
public String getUserPassword(){
        return userPassword;
}
public void setUserPassword(String userPassword){
        this.userPassword=userPassword;
}
@Column(nullable=false)
public int getUserType(){
        return userType;
}
public void setUserType(int userType){
        this.userType=userType;
}
```

```
@OneToMany(mappedBy="user", cascade=CascadeType.ALL,
fetch=FetchType.LAZY)
@OrderBy(value="fileID ASC")
public Set<File> getFiles(){
    return files;
}
......
}
```

- cascade的值可选择CascadeType.PERSIST(级联新建)、CascadeType.REMOVE(级联删除)、 CascadeType.REFRESH(级联刷新)、 CascadeType.MERGE(级联更新)中的一个或多个，CascadeType.ALL表示选择全部选项；
- fetch可选FetchType.EAGER(关系类File在主类加载的同时加载)、FetchType.*LAZY*（关系类在被访问时才加载，默认值）
- @OrderBy(value="fileID ASC")：在加载File时，按fileID的升序排序

File

```
……
@Entity
@Table(name="Filetbl")
public class File implements Serializable {
    private int fileID;
    private String fileName;
    private String fileSubject;
    private String filePath;
    private String fileType;
    //无fileOwner属性
    private User user;

    @Id
    @GeneratedValue
    public int getFileID(){
        return fileID;
    }
    public void setFileID(int fileID){
        this.fileID=fileID;
    }
    ……
```

```
    @ManyToOne(optional=true)
    @JoinColumn(name="fileOwner")
    public User getUser(){
        return user;
    }
    //optional，关系是否必须存在，即是否允许一端为NULL
    //@JoinColumn(name="fileOwner")，指定Filetbl表的fileOwner列作为外键与User的主键关联
    public void setUser(User user) {
        this.user=user;
    }
}
```

- EJB项目：
  - User,File：实体类
  - persistence.xml：实体Bean的配置文件（与Account相同）
  - UserDAORemote：使用实体Bean的会话Bean的业务接口
  - UserDAO：使用实体Bean的会话Bean的实现类
- Web项目：
  - UserFileServlet

UserDAORemote

```
……
@remote
public interface UserDAORemote {
    public boolean insertUser(String userID, String
    userName, String userMail, String userPassword,
    int userType);
    public String getUserNameByID(String userID);
    public boolean updateUser(User user);
    public User getUserByID(String userID);
    public List getUserList();
}
```

UserDAOBean

```
……
@Stateless
public class UserDAO implements UserDAORemote {
    @PersistenceContext
    protected EntityManager em;

    public User getUserByID(String userID) {
        User user = em.find(User.class, userID);
        return user;
    }
    public String getUserNameByID(String userID) {
        User user = em.find(User.class, userID);
        return user.getUserName();
    }
    public boolean updateUser(User user) {
        try{
            em.merge(user);
        }catch(Exception e){
            e.printStackTrace();
            return false;
        }
        return true;
    }
```

```
public boolean insertUser(String userID, String
userName, String userMail,String userPassword,
int userType) {
    try{
        User user = new User();
        user.setUserID(userID);
        user.setUserName(userName);
        user.setUserMail(userMail);
        user.setUserPassword(userPassword);
        user.setUserType(userType);
        em.persist(user);
    }catch(Exception e){
        e.printStackTrace();
        return false;
    }
    return true;
}
```

```
public List getUserList() {
    try{
        Query query = em.createQuery("from User u left join
        fetch u.files order by userid asc");
        List list =query.getResultList();
        em.clear();
        return list;
    }catch(Exception e){
        e.printStackTrace();
        return null;
    }
}
}
```

UserFileServlet

```
……
//更新User表（Name：Mary）
User user=dao.getUserByID(1234567890);
user.setName("Mary");
dao.updateUser(user);
```

```
//查询User表和File表
List list=dao.getUserList();
String userID=null;
for(int i=0; i<list.size();i++){
        User u=(User) list.get(i);
        if(u.getUserid()!=null) {
                if(!u.getUserid().equals(userID)) { //去除重复项
                        userID=u.getUserid();
                        out.println("用户UserID: "+userID);
                        Iterator iterator=(Iterator)
u.getFiles().iterator();
                        while(iterator.hasNext()){
                                File f=(File)iterator.next();
                                        out.println("拥有的文件
"+f.getFileName());
                        }
                }
        }
}
……
```

运行结果

- 用户ID：user1
- 拥有的文件 课程表.doc
- 拥有的文件 基金项目指南.doc
- 用户ID：user2

M:N关系（一）

- 如果中间表仅仅是做关联用的，仅有2个外键做联合主键，使用ManyToMany
  - 不用写中间表的实体类，只需要写出两张主表的实体类即可
  - 并且不需要手动创建中间表，JPA会根据配置自动创建
  - 示例：Hibernate.ppt

M:N关系（二）

- 如果中间表不仅仅是做关联用的，还包含了其他字段信息
  - 需要写三个实体类
  - 解决方案1：多对多的关系拆分为两个一对多
    - 与User-Files示例类似
    - 易理解
  - 解决方案2：两张主表实体类之间的关系为多对多；中间表实体类与两张主表的关系为多对一
    - 示例：Hibernate.ppt

createNativeQuery原生查询

- JPQL，基本上能符合绝大多数查询需求
- 当JPQL不能满足需求时，可使用原生SQL查询
  - Query createNativeQuery(String sql)
- 例如：

```
Query query = em.createNativeQuery("SELECT *
FROM t_user WHERE ID_NUM=?");
 query.setParameter(1, "64156519760630423 1");
    List<User>
userlist=(List<User>)query.getResultList();
```