

---

# Hibernate ORM

2023 03 20

# 參考資料與目錄

---

- 教學網站

- <https://www.youtube.com/channel/UCYeDPubBiFCZXIOgGYoyADw>

- 相關書籍

- Java Persistence with Hibernate

- Christian Bauer, Gavin King

- <http://www.amazon.com/Java-Persistence-Hibernate-Christian-Bauer/dp/1932394885?tag=javamysqlanta-20>

- 目錄

一、Object/Relational Mapping	p003
二、Hibernate核心類別與介面	p037
三、撰寫Hibernate程式	p063
四、延遲加載	p113
五、Hibernate Association	p127
六、Hibernate Query Language(HQL)	p179
七、附錄	p201

---

# 一、Object/Relational Mapping

# JDBC的利弊

---

- 優點

- 容易撰寫
- 處理大量資料時依然很有效率
- 適合中小型應用系統
- 存取資料庫的Java語法單純，易學易懂

- 缺點

- 必須自行JOIN多個表格內的資料，這使得同時讀取多個表格內的資料變的複雜
- 程式需要大量的額外敘述處理記錄內的欄位
  - 顯示資料時需要自行將表格內的紀錄轉換為物件
  - 儲存資料時需要自行將物件的屬性搬移到表格的欄位
- 存取表格所需的SQL敘述與使用的資料庫有強烈的相依性

# 何謂ORM (Object Relational Mapping)

---

- 物件關係映射是一種程式設計技術，它可在關聯式資料庫(Relational database)與物件導向語言(Object-Oriented Programming)之間進行資料轉換。
- 期望能以存取物件的方式來操作表格內的記錄
  - 程式執行Hibernate提供之Session介面的save(obj)方法，就能將參數obj寫入資料庫成為表格內的一筆新記錄
  - 程式執行上面之Session介面的delete(obj)方法就能刪除表格中與參數obj所對應的記錄
  - 經由類似的做法進行記錄的查詢、修改、結合多個表格與其他操作

# 知名的Java OR Mapping技術

---

- TopLink(Oracle)
- Entity Bean(EJB 2.x)
- Java Data Object(JDO, Sun)
- Object/Relational Bridge(Apache)
- **Hibernate**
- iBATIS (原稱為myBATIS)
- Java Persistence API(Sun, EJB 3.x)
  - Hibernate 4.3 series 開始支援JPA 2.1
  - Hibernate 5.3 series 開始支援JPA 2.2
  - Hibernate 5.5 series 開始支援JPA 3.0
- Spring Framework提供簡單的OR Mapping

# Hibernate ORM

---

- 由Gavin King於2001年創建之開放原始碼的永續儲存框架，為Java程式提供功能強大的永續儲存服務。
- 它提供物件與關聯式資料庫的對應關係(ORM)
  - 由Java類別對應到資料庫內的表格(Table)
  - 由Java物件對應到表格的紀錄(Record)
  - 由Java類別的屬性對應到紀錄的欄位(Field)
  - 提供資料查詢與資料存取的功能
  - 經由一些巧妙的設計，Hibernate可大量節省存取資料庫所需時間
  - 由於幾乎不使用SQL敘述，因此Hibernate程式與特定廠牌資料庫無關
    - 各種資料庫都有各自的方言，如T-SQL, PL/SQL
    - Hibernate遮蔽存取資料庫所需之細節，提供簡單、一致的操作介面，大幅提高程式師的生產力。

# Hibernate ORM的優點

---

- 根據Java類別的註釋說明對應關係
  - 早期用XML映射檔說明對應關係，現除舊系統外已無人使用
- 提供簡單且功能強大的API存取資料庫內的資料
- Hibernate可依類別內的註釋自動建立表格。如果表格結構變動僅需修改對應的註釋，Hibernate會自動產生修改表格的SQL敘述。
- 存取資料庫的Java程式碼與資料庫的廠牌無關
- 提供容易使用的交易管理
- 由於大量減少程式碼因此可以加快應用系統的開發速度



# 替專案引入Hibernate API

---

- 在專案的pom.xml中加入下列標籤就可為專案引入一組使用Hibernate API所需的jar檔：

```
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-core</artifactId>  
  <version>5.6.14.Final</version>  
</dependency>
```

# 使用範例程式前的注意事項

---

- 確認資料庫已安裝成功並已經啟動
- 確認相關的JDBC驅動程式已安裝在%TOMCAT\_HOME%的lib資料夾內
- 建立專案使用的資料庫JSPDB
- 匯入範例專案
- 修改專案中存取資料庫的帳號與密碼，範例專案使用的預設帳號與密碼：
  - MySQL : root / Do!ng123
  - SQL Server : sa / sa123456
  - 可使用Eclipse的Search > File 功能進行置換

# 比較Hibernate與JDBC兩種技術

---

- 比較兩種技術在存取資料庫之基本功能上的差異：
  - － 儲存一筆紀錄
  - － 查詢一筆紀錄
  - － 刪除一筆紀錄
  - － 修改一筆紀錄
  - － 查詢多筆紀錄

```
public Object save(Member mem){  
    Session session =  
        factory.getCurrentSession();  
    session.save(mem);  
    return mem;  
}
```

```
public Object save(Member mem) {  
    String sql = "INSERT INTO ch04_MemberExample "  
        + "(account, password, name, phoneNo, birthday, "  
        + " experience, registerTime) VALUES( ?, ?, ?, ?, ?, ?, ?)";  
    int n = 0;  
    try {  
        DataSource ds = (DataSource)  
            ctx.lookup("java:comp/env/jdbc/MemberDataBase");  
        try (  
            Connection conn = ds.getConnection();  
            PreparedStatement stmt = conn.prepareStatement(sql);  
        ) {  
            stmt.setString(1, mem.getUserId());  
            stmt.setString(2, mem.getPassword());  
            stmt.setString(3, mem.getName());  
            stmt.setString(4, mem.getPhoneNo());  
            stmt.setDate(5, mem.getBirthDay());  
            stmt.setInt(6, mem.getExperience());  
            stmt.setTimestamp(7, mem.getRegisterTime());  
            n = stmt.executeUpdate();  
        }  
    } catch (Exception e) {  
        throw new RecordNotFoundException(e);  
    }  
    return n;  
}
```

ch01.ex00.dao.impl.MemberDaoImpl\_Hibernate.java

ch01.ex00.dao.impl.MemberDaoImpl\_Jdbc.java

```
public Member findById(Integer id) {
    Member member = null;
    Session session =
        factory.getCurrentSession();
    member = session.get(Member.class, id);
    return member;
}
```

```
public Member findById(Integer id) {
    Member member = null;
    String sql =
        "SELECT * FROM ch04_MemberExample WHERE id = ?";
    try {
        DataSource ds = (DataSource)
            ctx.lookup("java:comp/env/jdbc/MemberDataBase");
        try (
            Connection conn = ds.getConnection();
            PreparedStatement stmt =
                conn.prepareStatement(sql);
        ) {
            stmt.setInt(1, id);
            try (ResultSet rset = stmt.executeQuery();) {
                if (rset.next()) {
                    member = new Member(rset.getInt("id"),
                        rset.getString("account"),
                        rset.getString("password"),
                        rset.getString("name"),
                        rset.getString("PhoneNo"),
                        rset.getInt("experience"),
                        rset.getDate("birthday"),
                        rset.getTimestamp("registerTime")
                    );
                }
            }
        }
    } catch (Exception e) {
        throw new RecordNotFoundException(e);
    }
    return member;
}
```

# 刪除一筆紀錄

# Hibernate vs JDBC

```
@Override
public void delete(Integer id) {
    Session session =
        factory.getCurrentSession();
    Member mem0 = findById(id);
    session.delete(mem0);
}
```

```
public void delete(Integer id) {
    String sql = "DELETE FROM ch04_MemberExample
        WHERE id = ?";
    try {
        DataSource ds = (DataSource)

            ctx.lookup("java:comp/env/jdbc/MemberDataBase");
        try (
            Connection conn = ds.getConnection();
            PreparedStatement stmt =
                conn.prepareStatement(sql);
        ) {
            stmt.setInt(1, id);
            int count = stmt.executeUpdate();

            if (count == 0) {
                throw new RecordNotFoundException
                    ("無法刪除紀錄或該筆紀錄不存在");
            }
        }
    } catch (Exception e) {
        throw new RecordNotFoundException(e);
    }
    return;
}
```

```
public void update(Member mem) {  
    Session session =  
        factory.getCurrentSession();  
    session.saveOrUpdate("Member", mem);  
}
```

```
public void update(Member mem) {  
    String sql = "UPDATE ch04_MemberExample SET  
        account=?, password=?, name=?, phoneNo=?,  
        experience=?, birthday=? where id = ?";  
    try {  
        DataSource ds = (DataSource)  
            ctx.lookup("java:comp/env/jdbc/MemberDataBase");  
        try (  
            Connection conn = ds.getConnection();  
            PreparedStatement stmt =  
                conn.prepareStatement(sql);  
        ) {  
            stmt.setString(1, mem.getUserId());  
            stmt.setString(2, mem.getPassword());  
            stmt.setString(3, mem.getName());  
            stmt.setString(4, mem.getPhoneNo());  
            stmt.setInt(5, mem.getExperience());  
            stmt.setDate(6, mem.getBirthday());  
            stmt.setInt(7, mem.getId());  
            int count = stmt.executeUpdate();  
            if (count == 0) {  
                throw new RecordNotFoundException("無法更新紀錄或  
                該筆紀錄不存在");  
            }  
        }  
    } catch (Exception e) {  
        throw new RecordNotFoundException(e);  
    }  
    return;  
}
```

```
public List<Member> findAll() {  
    String hql = "FROM Member";  
    List<Member> allMembers = null;  
    Session session =  
        factory.getCurrentSession();  
    allMembers =  
        session.createQuery(hql, Member.class)  
            .getResultList();  
    return allMembers;  
}
```

```
public List<Member> findAll() {  
    List<Member> allMembers = new ArrayList<Member>();  
    String sql = "SELECT * FROM ch04_MemberExample";  
    try {  
        DataSource ds = (DataSource)  
            ctx.lookup("java:comp/env/jdbc/MemberDataBase");  
        try {  
            Connection conn = ds.getConnection();  
            PreparedStatement stmt =  
                conn.prepareStatement(sql);  
            ResultSet rs = stmt.executeQuery();  
        } {  
            Member mem = null;  
            while (rs.next()) {  
                mem = new Member(  
                    rs.getInt("id"),  
                    rs.getString("account"),  
                    rs.getString("password"),  
                    rs.getString("name"),  
                    rs.getString("PhoneNo"),  
                    rs.getInt("experience"),  
                    rs.getDate("birthday"),  
                    rs.getTimestamp("RegisterTime")  
                );  
                allMembers.add(mem);  
            }  
        }  
    } catch (Exception e) {  
        throw new RecordNotFoundException(e);  
    }  
    return allMembers;  
}
```

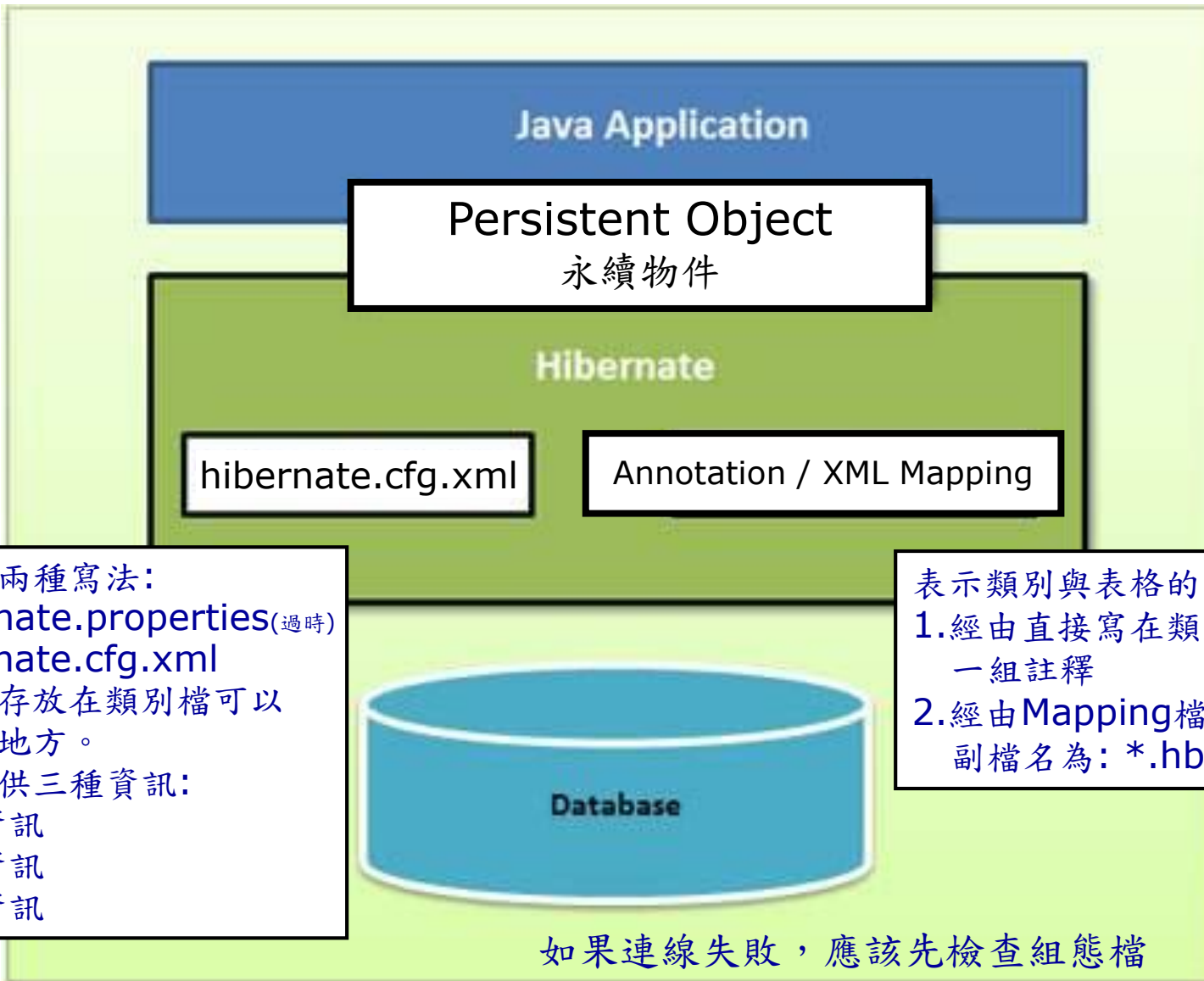


# 不適合使用Hibernate的時機

---

- 大量的資料異動(如數萬筆以上的紀錄)，應該改用 Stored Procedure以增進效能。
- 長達一頁以上的SQL敘述不適合由Hibernate完成，應改用其他方式(如JDBC程式)完成。

# Hibernate架構圖



# Hibernate組態檔

---

- 組態檔有兩種寫法：
  - Xml格式，習慣上檔案命名為**hibernate.cfg.xml**
  - **hibernate.properties**(過時，不需要花時間研究)
  - 它們必須存放在類別檔可以被找到的地方
    - Maven-based專案應放在src/main/resources資料夾下
- 組態檔提供三種資訊：
  - 連線資訊
    - 包括連線類別、連線字串、帳號、密碼與連線池的相關設定等
  - 映射資訊
    - 在永續類別內使用註釋說明映射資訊，或
    - 每個永續類別使用一個Mapping檔(\*.hbm.xml)說明映射資訊(較舊)
  - 進階資訊
    - 說明使用何種廠牌資料庫以及版本(SQL方言)
      - 此項資訊與資料庫有關，故經常與連線資訊寫在一起
    - 是否要顯示系統產生的SQL敘述與是否要經過編排
    - 是否要在啟動Hibernate時自動產生資料庫表格
      - 由Hibernate組態檔內的<hibernate.hbm2ddl.auto>標籤設定

# 連線資訊 (透過JDBC API連結資料庫)

<!-- 1. SQL方言與提供連結資料庫所需資訊(透過JDBC API連結資料庫)之設定方式 -->

<!-- **Database**是**SQL Server**, 設定 SQL方言、JDBC驅動程式的主類別、連線的URL、帳號與密碼, -->

<property name="hibernate.dialect">org.hibernate.dialect.SQLServer2012Dialect</property>

<property name="connection.driver\_class">com.microsoft.sqlserver.jdbc.SQLServerDriver</property>

<property name="connection.url">jdbc:sqlserver://127.0.0.1:1433;DatabaseName=JSPDB</property>

<property name="connection.username">sa</property>

<property name="connection.password">sa123456</property>

<!-- 1. SQL方言與提供連結資料庫所需資訊(透過JDBC API連結資料庫)之設定方式 -->

<!-- **Database**是**MySQL**, 設定 SQL方言、JDBC驅動程式的主類別、連線的URL、帳號與密碼, -->

<property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">Do!ng123</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>

<property name="connection.url"> jdbc:mysql://localhost:3306/jspdb?**useSSL=false&**  
**useUnicode=yes&**;**characterEncoding=utf8&**;**serverTimezone=Asia/Taipei**  
**&allowPublicKeyRetrieval=true**  
</property>

由於資料太長不利排版，所以講義  
分為數列印出，使用時要寫成一列

參考: \src\main\resources\hibernate.cfg.xml

# 映射資訊

---

- 經由類別內的註釋(Annotation)
  - 通知Hibernate哪些在類別含有說明映射資訊的註釋  
**<mapping class="ch01.model.Department"/>**  
**<mapping class="ch01.model.Employee"/>**
- 經由映射檔(舊的用法，已經不流行)
  - 通知Hibernate哪些映射檔含有說明映射資訊的XML標籤  
**<mapping resource="member.hbm.xml" ></mapping>**
  - 此檔應該放在: src/main/resources之下(maven-based)

- 自動新建或更新資料庫內表格的Schema

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

- **create**：新建SessionFactory物件時，會刪除現有的表格，然後根據類別內的註釋重新產生表格，表格內原有資料會遺失。建立應用系統的初始資料應該採用此選項。
- **create-drop**：新建SessionFactory物件時，根據類別的映射資訊重新產生表格，關閉SessionFactory時，自動刪除表格。
- **update**：新建SessionFactory物件時，若表格不存在就新建表格，如果表格存在，根據類別內的註釋自動更新表格結構，如果類別加入新的實例變數與對應的Getter/Setter，Hibernate會自動插入新的欄位以改變表格結構；如果刪除類別的實例變數與Getter/Setter表格內原有的欄位與其內資料仍然存在而不會刪除它們，僅不再使用這些移除的欄位。此為開發階段最常用的選項。

- **validate** : 新建SessionFactory物件時根據類別的映射資訊驗證表格結構，只會和資料庫中的表格進行比較，不會創建新表格。若表格結構不同將會丟出例外。
- **none** : Hibernate不對表格做任何處理。

- 於主控台顯示Hibernate自動產生的SQL敘述  
`<property name="show_sql">true</property>`
- 採編排、內縮的方式顯示Hibernate自動產生的SQL敘述  
`<property name="format_sql">true</property>`
- 新建Session物件會連結到何種環境  
`<property name="hibernate.current_session_context_class">thread</property>`
  - 當經由sessionFactory.getCurrentSession()方法新建一個Session物件時，sessionFactory會將Session物件『綁到』某個環境。當此標籤值為『thread』時，sessionFactory會將Session物件『綁到』目前正在執行的執行緒。稍後呼叫sessionFactory.getCurrentSession()會返回先前產生、綁在執行緒上的Session物件。
  - 其他的可能值包括'jta'，sessionFactory會將Session物件『綁到』目前正在執行的JTA(Java Transaction API)交易，前提是你的應用程式伺服器有支援JTA。
  - JTA：定義了交易管理者(Transaction Manager)、資源管理者(Resource Manager)、應用程式伺服器與應用程式之間的標準介面。有了這樣的介面，需要撰寫分散式交易的各方程式就有了一致性的標準，有利於各方程式的開發。



# 練習一

---

- WebAppLab01 專案為一個使用Servlet/JSP與JDBC開發的網路應用程式，可對Member表格進行資料的新增、刪除、查詢、修改等操作，現要以Hibernate技術取代JDBC技術而完成相同的工作。
- 本練習需要：
  - 匯入WebAppLab01專案
  - 打開專案的pom.xml，為其加入相關的<dependency>
  - 替專案加入hibernate.cfg.xml，修改組態檔內的連線資訊
- 測試前必須先依所用的資料庫執行建立初始表格所需要的SQL檔：
  - src\main\resources\MySQL\_Init.sql (MySQL), 或
  - src\main\resources\SQLServer\_Init.sql (SQLServer)

# 經由註釋(Annotation)提供映射資訊

---

- 現今流行的做法，取代舊式Mapping檔
  - 必須搭配JDK 5.0與Hibernate 3.x版才能使用註釋
  - 映射資訊：類別與哪個表格對應，類別內的屬性與表格內的哪個欄位對應，表格欄位的名稱、型態與長度，類別間的關聯。當類別編譯後這些註釋會存入.class檔。
- 相關套件
  - javax.persistence.\*：EJB 3.0標準
    - 挑選屬於此套件下的註釋
  - org.hibernate.\*：Hibernate特有功能
    - 挑選屬於此套件下的類別與介面
- 類別內編寫註釋的兩種方式：
  - 寫在實例變數之前(Field Access)，較優，建議使用做法。
  - 寫在**Getter**方法之前(Property Access)
  - 只能二選一，不能混用

# Employee類別

■

```
package ch01.model;

import java.util.Date;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity(name = "ch01_Employee")
@Table(name = "ch01_Employee_Table")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Integer id;
```

# Employee類別

..

```
@Column(name = "employee_Id", columnDefinition = "VARCHAR(10) NOT NULL")
String employeeId;
```

```
String name;
```

```
Integer salary;
```

```
@Column(columnDefinition = "DECIMAL(5,1)")
```

```
Double weight;
```

```
// @Temporal只能用在java.util.Date, @Temporal(TemporalType.DATE)表示刪除時分秒，  
// 僅保留年月日。
```

```
@Temporal(TemporalType.DATE)
```

```
Date birthday;
```

```
@ManyToOne(cascade=CascadeType.ALL)
```

```
@JoinColumn(name="dept_id", foreignKey=@ForeignKey(name = "fk_emp_dep"))
```

```
@JoinColumn(name="dept_id")
```

```
Department dept;
```

```
public Employee() {  
}
```

# Employee類別

...

```
public Employee(Integer id, String employeeId, String name,
    Integer salary, Double weight, Date birthday, Department dept) {
    this.id = id;
    this.employeeId = employeeId;
    this.name = name;
    this.salary = salary;
    this.weight = weight;
    this.birthday = birthday;
    this.dept = dept;
}

public Employee(Integer id) {
    this.id = id;
}

public Integer getId() {
    return id;
}

public void setPk(Integer id) {
    this.id = id;
}
```

# Employee類別

....

```
public String getEmployeeId() {  
    return employeeId;  
}  
public void setEmployeeId(String employeeId) {  
    this.employeeId = employeeId;  
}  
public String getName() {  
    return name;  
}  
public void setName(String name) {  
    this.name = name;  
}  
public int getSalary() {  
    return salary;  
}  
public void setSalary(int salary) {  
    this.salary = salary;  
}  
// 省略剩餘的Getter/Setter
```

# 註釋說明一：

- @Entity

- 標示本類別是受到Hibernate控管的永續類別(凡資料庫中有表格與之對應的類別稱為永續類別)，此註釋必須寫在類別名稱之前。
  - name屬性指定Entity的名稱，預設的Entity Name為類別名稱。
- @Entity與@Id為永續類別一定要撰寫的兩個註釋
- 該類別一定要有預設建構子

- @Table

- 說明本類別對應之表格的相關屬性，與@Entity皆為類別層級(Class Level)註釋
- 相關屬性
  - name="ch01\_Employee\_Table" 指定表格名稱，省略此屬性時表格名稱就是類別名稱
  - catalog="JSPDB" 指定資料庫名稱，省略此屬性時將使用連線字串上的資料庫名稱
  - uniqueConstraints= {@UniqueConstraint(columnNames={"employee\_Id", "empName"})}) 指定表格的限制條件。employee\_Id與empName必須是表格內欄位名稱。此條件為employee\_Id與empName的組合不可以重複。

- @Id

- 本類別中具有唯一性的欄位，對應表格中的主要鍵。若註釋@Id寫在某個實例變數前，則其他對應表格欄位的註釋都必須寫在實例變數前。若註釋@Id寫在某個getter前，則其他用以對應表格欄位的註釋都必須寫在getter前。

## 註釋說明二:

---

- `@GeneratedValue(strategy = GenerationType.IDENTITY)`
  - 說明表格中的主要鍵的生成方式，
  - `GenerationType.AUTO`表示由Hibernate依照所使用的資料庫自動挑選一種主鍵生成策略以產生主鍵值。
  - `GenerationType.SEQUENCE`表示由SEQUENCE表格產生主鍵值
  - `GenerationType.IDENTITY`表示由底層的資料庫提供的自增欄位機制產生主鍵值。
- `@Column(columnDefinition="Date", name = "birthday" )`
  - 指定欄位的名稱(name屬性)與欄位的其他性質，如欄位的型態、長度、NOT NULL等(columnDefinition)
  - 可省略，只有在欄位名稱與屬性名稱不同時使用。
  - `columnDefinition="Date"`：說明欄位的型態。
  - `columnDefinition="VARCHAR(50) NOT NULL"`：說明欄位的型態為VARCHAR。
- `@Temporal(TemporalType.DATE)`
  - 顯示年、月、日而不顯示時、分、秒。



## 註釋說明三：

---

- **@ManyToOne(cascade=CascadeType.ALL)**
  - 說明本表格(Employee)與對照表格(Department)為多對一的關係
  - **cascade=CascadeType.ALL**：無論儲存、合併、更新或刪除，一併對被關聯的物件作出一樣操作。
- **@OneToMany**
  - 說明本表格與對照表格為一對多的關係
- **@OneToOne**
  - 說明本表格與對照表格為一對一的關係
- **@ManyToMany**
  - 說明本表格與對照表格為多對多的關係
- **@Transient**
  - 預設的情況下，永續類別內未使用**static**與**transient**修飾的所有實例變數都會對應表格內一個欄位，除非使用**@Transient**標示。你可以在實例變數之前或對應的**getter**之前使用**@Transient**說明該實例變數不要與表格的欄位對應。

# 在組態檔(hibernate.cfg.xml)中說明永續類別

---

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- 省略許多設定 -->
    .....
    <!-- 省略許多設定 -->

    <!-- 通知Hibernate哪些類別檔含有說明映射資訊的註釋 -->
    <mapping class="ch01.model.Department"/>
    <mapping class="ch01.model.Employee"/>
  </session-factory>
</hibernate-configuration>
```

## 練習二

---

- 替 WebAppLab01 專案加入映射資訊
  - 在專案內的所有永續類別內加入適當的註釋
  - 在專案的hibernate.cfg.xml加入含有映射資訊的永續類別



---

## 二、Hibernate核心類別與介面

# 編寫Hibernate程式的通則

---

1. 建構一個org.hibernate.SessionFactory(介面的)物件
2. 由SessionFactory物件得到一個org.hibernate.Session(介面的)物件
3. 執行session物件的beginTransaction()開啟交易，同時取得交易物件(org.hibernate.Transaction)。
4. 使用session物件提供的方法進行表格紀錄的增、刪、改、查：
  - session.save(obj)儲存一個新的物件
  - session.update(obj)更新一個舊有物件
  - session.merge(obj)更新一個舊有物件
  - session.saveOrUpdate(obj)新增或更新一個物件
  - session.delete(obj)刪除一個物件
  - session.get(Class, id)依照主鍵值讀取一個物件
  - session.createQuery(hql)建立一個可存取表格的Query物件
5. 依照執行結果呼叫Transaction物件的commit()或rollback()。
6. 關閉Session物件。
7. 關閉SessionFactory物件。

# 建立SessionFactory物件 (限Hibernate 5.x)

```
private static SessionFactory buildSessionFactory() {  
    try {  
        // Hibernate 5.x 的寫法：在組態檔內定義永續類別  
        StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()  
            .configure("hibernate.cfg.xml").build();  
        Metadata metadata = new MetadataSources(standardRegistry).getMetadataBuilder().build();  
        SessionFactory sessionFactory = metadata.getSessionFactoryBuilder().build();  
        return sessionFactory;  
    } catch (Throwable ex) {  
        System.err.println("新建SessionFactory失敗：" + ex.getMessage());  
        throw new ExceptionInInitializerError(ex);  
    }  
}
```

- `configure("hibernate.cfg.xml");`
  - 相對於類別路徑的根目錄。在Eclipse的Java Build Path內，source選項定義的資料夾都是類別路徑的根目錄。
- Hibernate 5.0以前的寫法多有變化，請參考附錄。

# SessionFactory 介面

---

- 它儲存編譯過的映射資訊與Hibernate在執行時所產生的SQL敘述與表格、關聯的Metadata。
- 此介面的主要功能為**管理**org.hibernate.**Session**物件。使用Hibernate技術的Java程式每次存取表格資料前都必須先取得Session物件。
- 一個Database對應一個SessionFactory物件。
- 實作SessionFactory介面的類別是ThreadSafe，所以允許有多個執行緒共用同一個SessionFactory物件取得Session物件。



# 建立SessionFactory的時機

---

- 建立SessionFactory物件耗費許多系統資源與時間，因此建立此物件的時機應有詳細的規劃。
- Java SE應用程式
  - 只要準備存取表格資料前便可建立SessionFactory物件。可考慮利用靜態區塊建立SessionFactory物件。
  - 程式將使用輔助類別HibernateUtils.java的靜態方法建立此物件，並由此類別的靜態方法getSessionFactory()方法取得SessionFactory物件。
- Web應用程式
  - 為網路應用程式設計並註冊一個實作ServletContextListener介面的類別，Override該類別的contextInitialized()方法，在該方法中呼叫HibernateUtils.java的靜態方法建立此物件。
  - 當Tomcat啟動後對我們的網路應用系統進行初始化時便會建立SessionFactory物件，供系統內的其他元件使用。

# Session 介面

---

- Hibernate程式每次存取資料庫之前必須先取得一個Session物件，此物件配置一個可以連上資料庫的實體連線。
- 程式必須經由Session物件開啟交易(beginTransaction())，才能進行表格資料的增刪改查。Session介面提供許多方法進行儲存/查詢/修改/刪除物件所對應的表格紀錄。
- 程式可以透過SessionFactory物件的openSession()與getCurrentSession()取得Session物件。
- Session物件不是ThreadSafe，因此不可以在多執行緒的環境下使用，即一個Session物件只能給一個執行緒使用。使用Session物件應該把握的原則：
  - Session型態的變數必須是區域變數。
  - 需要使用它時才開啟，用完後立刻關閉。註：此Session物件與Servlet內的HttpSession沒有任何關係

# Session物件的關閉

---

- 當程式不再需要存取資料庫時，一定要執行Session介面的close()方法以關閉Session物件。
- 關閉Session物件並不一定等於關閉資料庫連線，如果Hibernate在提供連接池(Connection pool)的環境下執行，Session的close()方法會將連線物件java.sql.Connection還回連接池而不會關閉它。

# openSession()

---

- 每次程式呼叫SessionFactory介面的openSession()方法都會得到一個全新的Session物件。
- 當程式不需要存取資料庫時必須執行session.close()方法關閉Session物件。
- 不需要做任何組態設定就可以使用openSession()方法。
- 下面的程式片段可以看出每次執行openSession()都會建立一個新的Session物件：

```
Session session1 = HibernateUtils.getSessionFactory().openSession();
Session session2 = HibernateUtils.getSessionFactory().openSession();
System.out.println(session1);
System.out.println(session2);
boolean isEqual = session1.equals(session2);
System.out.println("由SessionFactory的openSession()取出的兩個Session物件是否相等:"
    + isEqual);
```

ch02.OpenSessionDemo.java

# getCurrentSession()

---

- 當存取資料庫的程式第一次執行SessionFactory介面的getCurrentSession()方法時會開啟一個全新的Session物件，然後此Session物件就會綁定在Hibernate控管的環境內。
- 往後程式每次執行getCurrentSession()方法會得到同一個Session物件。由於交易都需要在同一個Session物件中進行，因此當程式的交易橫跨多個DAO類別時一定要以此方法取得Session物件。
- 程式不需要關閉這樣的Session物件，因為當程式執行交易的commit()或rollback()方法讓交易結束後，會自動關閉Session物件。
  - － 組態檔(hibernate.cfg.xml)必須加入下列標籤，否則程式會丟出No CurrentSessionContext configured!的錯誤訊息。

`<property name="hibernate.current_session_context_class">thread</property>`

# GetCurrentSessionDemo.java

---

```
package ch02;

import org.hibernate.Session;

import ch00.HibernateUtils;

public class GetCurrentSessionDemo {

    public static void main(String[] args) {
        Session session1 = HibernateUtils.getSessionFactory().getCurrentSession();
        Session session2 = HibernateUtils.getSessionFactory().getCurrentSession();
        System.out.println(session1);
        System.out.println(session2);
        boolean isEqual = session1.equals(session2);
        System.out.println("由SessionFactory的getCurrentSession()取出的兩個Session物件是否相等:"
            + isEqual);
    }
}
```

# openSession() vs getCurrentSession()

	openSession()	getCurrentSession()
何時產生Session物件	永遠產生新的Session物件	在Hibernate控管的環境內尋找本程式對應的Session物件，如果找到就取出使用，否則建立一個新的Session物件並放入控管的環境。
何時關閉Session物件	程式需要自行執行session的close()方法。	當程式執行交易的commit()或rollback()方法讓交易結束後，會自動關閉Session物件。
跨多個DAO類別進行的交易	不能用此方法產生的Session物件進行跨多個DAO類別進行的交易	必須用此方法產生的Session物件進行跨多個DAO類別進行的交易
組態設定	不需要任何組態設定	必須在組態檔中加入 <current_session_context_class> 標籤

# Transaction 介面

---

- Transaction 介面是Hibernate提供的資料庫交易 (Transaction) 介面，它遮蔽資料庫底層真正進行交易之機制間的差異，提供一致的程式編寫方式，讓程式很容易定出交易的界線，有助於同一個應用系統在不同的Java EE容器移植。
- 一個交易中對多個表格的異動(增刪改查)一定要在同一個session物件下進行各個表格的異動。
- 在Hibernate程式中，所有存取資料庫的方法都必須在交易內進行。即要先開啟Transaction才進行資料庫的存取。如果成功存取資料進行交易的commit；如果存取資料失敗則進行交易的rollback。



# EmployeeServiceImpl.java 程式片段

---

@Override

```
public Object save(Employee emp) {
    Session session = factory.getCurrentSession();
    Transaction tx = null;
    Object key = null;
    try {
        tx = session.beginTransaction();
        String depName = emp.getDept().getDepName();
        if ( depName != null ) {
            Department dept = departmentDao.findByName(depName);
            if (dept != null) {
                emp.setDept(dept);
            }
        }
        key = employeeDao.save(emp);
        tx.commit();
    } catch (Exception e) {
        if (tx != null) {
            tx.rollback();
        }
        e.printStackTrace();
        throw new RuntimeException(e.getMessage());
    }
    return key;
}
```

# EmployeeDaoImpl.java 程式片段

---

```
public Object save(Employee emp) {  
    Session session = factory.getCurrentSession();  
    Object key = session.save(emp);  
    return key;  
}
```

# 第一個完整的Hibernate程式

---

- `ch01.main.SaveEmployee.java`
  - 建構物件然後透過EmployeeService物件的save()方法儲存物件。
- `ch01.util.HibernateUtils.java`
  - 負責建立SessionFactory物件，經由它提供的靜態方法getSessionFactory()方法取得SessionFactory物件。
  - SessionFactory物件能夠產生Session物件。
- `ch01.model.Employee.java`
  - 永續類別，定義類別與表格及屬性與欄位的映射資訊。
- `ch01.model.service`套件下的EmployeeService.java
  - 介面，定義程式對Employee類別要完成的功能，系統需要的企業邏輯都由此介面定義。
- `ch01.model.service.impl.EmployeeServiceImpl.java`
  - 實作EmployeeService介面的類別，它要完成的工作都委託給ch01.model.dao.EmployeeDao介面。

# 第一個完整的Hibernate程式

---

- `ch01.model.EmployeeDao.java`
  - 介面，定義Employee物件的新增、刪除、查詢與修改的方法。
- `ch01.model.dao.impl.EmployeeDaoImpl.java`
  - 實作EmployeeDao介面的類別，用Hibernate技術完成對Employee物件之新增、刪除、查詢與修改等方法。

# 公用程式HibernateUtils.java

- 本程式的功能：建立與關閉SessionFactory物件。SessionFactory物件的重要功能：產生Session物件。

// 省略 package 與 import 敘述

```
public class HibernateUtils {  
    private static SessionFactory sessionFactory = buildSessionFactory();  
    private static SessionFactory buildSessionFactory() {  
        try {  
            // Hibernate 5.x 的寫法  
            StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()  
                .configure("hibernate.cfg.xml").build();  
            Metadata metadata = new MetadataSources(standardRegistry).getMetadataBuilder().build();  
            SessionFactory sessionFactory = metadata.getSessionFactoryBuilder().build();  
            return sessionFactory;  
        } catch (Throwable ex) {  
            System.err.println("新建SessionFactory失敗:" + ex.getMessage());  
            throw new ExceptionInInitializerError(ex);  
        }  
    }  
    // 外界呼叫此靜態方法以取得 SessionFactory物件  
    public static SessionFactory getSessionFactory() {  
        return sessionFactory;  
    }  
    // 外界呼叫此靜態方法關閉 SessionFactory物件  
    public static void close() {  
        getSessionFactory().close();  
    }  
}
```

SessionFactory介面提供兩個方法可傳回Session物件。當程式需要Session物件時，只需要呼叫SessionFactory介面：  
getCurrentSession()或  
openSession() 方法就可以得到Session物件。

# SaveEmployee.java (摘要)

---

```
public class SaveEmployee {  
    public static void main(String[] args) {  
        EmployeeService employeeService = new EmployeeServiceImpl();  
        Department dept1 = new Department(null, "行銷部");  
        Department dept2 = new Department(null, "工程部");  
        Department dept3 = new Department(null, "會計部");  
  
        Employee emp1 = new Employee(null, "A033", "劉麗芳", 56000, 57.6,  
                                     Date.valueOf("1980-1-5"), dept1);  
        employeeService.save(emp1);  
  
        Employee emp2 = new Employee(null, "A070", "葉美華", 45000, 66.7,  
                                     Date.valueOf("1987-8-9"), dept1);  
        employeeService.save(emp2);  
  
        Employee emp3 = new Employee(null, "A120", "林國忠", 37000, 64.0,  
                                     Date.valueOf("1992-6-18"), dept1);  
  
        employeeService.save(emp3);  
    }  
}
```

# SaveEmployee.java (摘要)

---

```
Employee emp4 = new Employee(null, "B501", "黃湘", 48000, 60.0,  
    Date.valueOf("1990-5-17"), dept2);  
  
employeeService.save(emp4);  
  
Employee emp5 = new Employee(null, "C702", "劉德佳", 43500, 62.0,  
    Date.valueOf("1997-5-2"), dept3);  
  
employeeService.save(emp5);  
  
Employee emp6 = new Employee(null, "C715", "林曉真", 55000, 68.7,  
    Date.valueOf("1988-12-12"), dept3);  
  
employeeService.save(emp6);  
  
HibernateUtils.getSessionFactory().close();  
}  
}
```

# EmployeeService.java 介面

---

```
package ch01.model.service;

import java.util.List;
import ch01.model.Employee;

public interface EmployeeService {
    // 新增一筆Employee物件到資料庫
    Object save(Employee emp);

    // 新增一筆Employee物件到資料庫
    void persist(Employee emp);

    // 經由Session介面的get()查詢資料庫內的紀錄
    Employee findById(Integer id);

    // 更新紀錄
    void update(Employee e);

    // 刪除紀錄
    void delete(Integer id);

    // 查詢所有紀錄
    List<Employee> findAll();

    void close();
}
```



# EmployeeServiceImpl.java

-1

```
package ch01.model.service.impl;  
// 摘要  
public class EmployeeServiceImpl implements EmployeeService {  
  
    EmployeeDao    employeeDao;  
    DepartmentDao  departmentDao;  
    SessionFactory factory;  
  
    public EmployeeServiceImpl() {  
        employeeDao = new EmployeeDaoImpl();  
        departmentDao = new DepartmentDaoImpl();  
        factory = HibernateUtils.getSessionFactory();  
    }  
}
```

# EmployeeServiceImpl#save()

---

```
public Object save(Employee emp) {
    Session session = factory.getCurrentSession();
    Transaction tx = null;
    Object key = null;
    try {
        tx = session.beginTransaction();
        String depName = emp.getDept().getDepName();
        if ( depName != null ) {
            Department dept = departmentDao.findByName(depName);
            if (dept != null) {
                emp.setDept(dept);
            }
        }
        key = employeeDao.save(emp);
        tx.commit();
    } catch (Exception e) {
        if (tx != null) {
            tx.rollback();
        }
        e.printStackTrace();
        throw new RuntimeException(e.getMessage());
    }
    return key;
}
```

# EmployeeDao.java 介面

---

```
package ch01.model.dao;

import java.util.List;
import ch01.model.Employee;

public interface EmployeeDao {

    // 新增一筆Employee物件到資料庫
    Object save(Employee emp);

    // 經由Session介面的get()查詢資料庫內的紀錄
    Employee findById(Integer id);

    // 更新紀錄
    void update(Employee e);

    // 刪除紀錄
    void delete(Integer id);

    // 查詢所有紀錄
    List<Employee> findAll();

    void close();

    void persist(Employee emp);
}
```

# EmployeeDaoImpl.java

-1

```
package ch01.model.dao.impl;
// 省略import敘述
public class EmployeeDaoImpl implements EmployeeDao {

    SessionFactory factory;

    public EmployeeDaoImpl() {
        factory = HibernateUtils.getSessionFactory();
    }

    @Override
    public void persist(Employee emp) {
        Session session = factory.getCurrentSession();
        session.persist(emp);
    }

    public Object save(Employee emp) {
        Session session = factory.getCurrentSession();
        Object key = session.save(emp);
        return key;
    }

    // 更新紀錄
    public void update(Employee employee) {
        Session session = factory.getCurrentSession();
        session.saveOrUpdate(employee);
    }
}
```

# EmployeeDaoImpl.java

-2

```
public void delete(Integer id) { // 刪除紀錄
    Session session = factory.getCurrentSession();
    Employee emp = findById(id);
    if (emp != null ) {
        emp.setEmployeeId(null);
        session.delete(emp);
    } else {
        throw new RuntimeException("紀錄不存在，無法刪除");
    }
}
public void close() {
    factory.close();
}
@Override
public Employee findById(Integer id) {
    Session session = factory.getCurrentSession();
    Employee emp = (Employee) session.get(Employee.class, id);
    return emp;
}
@Override // 查詢所有紀錄
public List<Employee> findAll() {
    Session session = factory.getCurrentSession();
    List<Employee> allEmployees = session.createQuery("FROM ch01_Employee", Employee.class)
                                           .getResultList();
    return allEmployees;
}
}
```

# 練習三

---

- 替WebAppLab01專案加入HibernateUtils.java

---

## 三、撰寫Hibernate程式

# 永續儲存的類別(Persistence Class)

---

- 在Hibernate中，與資料庫表格對應的類別稱為永續儲存的類別(簡稱永續類別)。永續類別的屬性對應表格的欄位，該類別的一個個不同物件對應表格的一筆筆紀錄。
- Hibernate參考永續類別的映射資訊(儲存在永續類別內的Annotation或映射檔)決定：
  - － 寫出資料到表格時，物件屬性要寫入哪個表格的哪個欄位；
  - － 由表格讀出資料時，建構何種類別的物件以存放讀出的資料。
- 不可以是final類別
  - － 因為Hibernate會動態產生子類別繼承這些POJO類別
- 成為永續類別的要件
  - － 類別前加上@Entity，類別OID欄位前加上@Id
  - － hibernate.cfg.xml 內加上<mapping class='...' />標籤



# POJO類別

---

- POJO: Plain Old Java Object。POJO類別不會實作某個框架規定的介面或繼承某個父類別，永續類別如果符合POJO類別的規範，存取資料庫內的紀錄Hibernate可以發揮最大的效益。
- POJO類別
  - 必須定義預設建構子。屬性不可定義為基本型態，必須是對應的包裝類別(Integer, Long, Float, Double...等)
  - 必須定義唯一識別的屬性(OID, 對應Primary Key)
    - 在一個Session中，同一類別的兩個物件若OID相同，將會對應到同一筆記錄，Hibernate不允許這種情形發生。不要挑與Domain相關的欄位作為Primary Key。身分證號碼就是一個與Domain相關的欄位。
  - 所有需要寫入表格的屬性之存取修飾字最好是private，且都要定義Getter與Setter方法。
  - 若分離物件需要經由網路傳送到其他電腦，則需實作java.io.Serializable介面

# POJO類別的範例

---

```
public class Employee {  
    private Integer employeeId;  
    private String firstName;  
    private String lastName;  
    private Integer salary;  
    public Employee() { }  
    public Employee(String fname, String lname, Integer salary) {  
        this.firstName = fname;  
        this.lastName = lname;  
        this.salary = salary;  
    }  
    public Integer getEmployeeId() {  
        return employeeId;  
    }  
    public void setEmployeeId(Integer employeeId) {  
        this.employeeId = employeeId;  
    }  
    public String getFirstName() {  
        return firstName;  
    }  
}
```

# POJO類別的範例

---

```
public void setFirstName(String firstName) {  
    this.firstName = firstName;  
}  
public String getLastName() {  
    return lastName;  
}  
public void setLastName(String lastName) {  
    this.lastName = lastName;  
}  
public Integer getSalary() {  
    return salary;  
}  
public void setSalary(Integer salary) {  
    this.salary = salary;  
}  
}
```

# 永續物件的生命週期

- 物件的生命週期

- Transient →

```
Employee emp = new Employee();  
emp.setName("劉麗芳");  
emp.setEmployeeId("A033");  
emp.setSalary(40000);  
emp.setWeight(50.5);  
emp.setHeight(180.5);  
emp.setBirthday(java.sql.Date.valueOf("1980-05-20"));  
Department dept = new Department(null, "行銷部");  
emp.setDept(dept);
```

```
Session session = HibernateUtils.getSessionFactory().openSession();  
Transaction tx = session.beginTransaction();
```

- Persistent →

```
session.save(emp);  
emp.setName("章軍亞");  
emp.setName("劉麗芳");
```

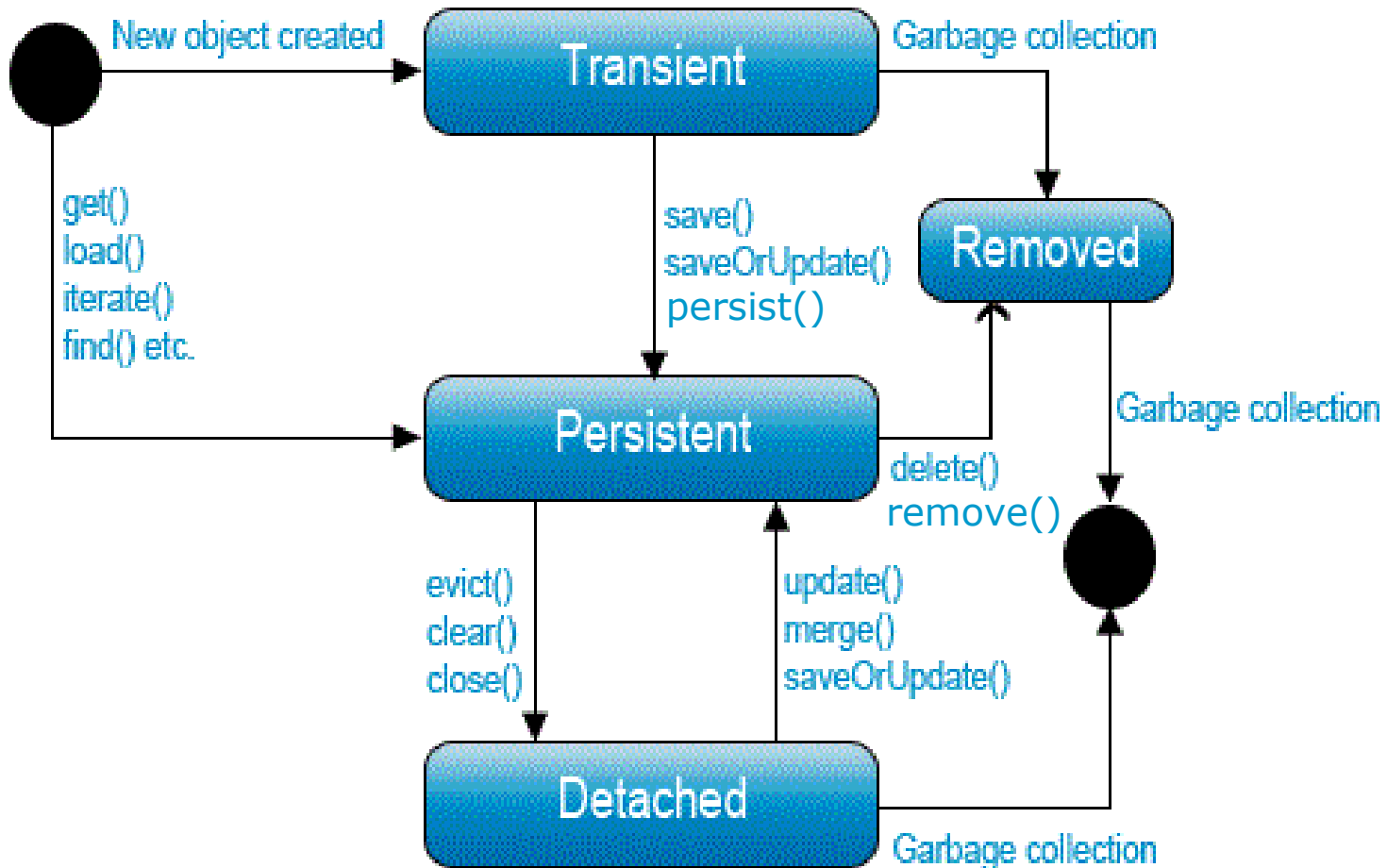
```
tx.commit();  
session.close();
```

- Detached →

```
emp.setName("黃妃虹");
```

- Removed

# 物件在Hibernate內生命週期的變化



# Transient(臨時)狀態

---

- 經由new運算子產生的任何物件，在未經Session物件的save()方法/persist()方法/saveOrUpdate()等方法將此物件儲存到表格前都處於臨時狀態。處於臨時狀態的物件稱為臨時物件。
  - 臨時物件沒有OID
  - 臨時物件在資料庫中沒有對應的紀錄
  - 改變臨時物件的屬性值不會影響表格內的任何紀錄。

# Persistent(永續)狀態

---

- 當臨時物件經由Session物件的save()方法/persist()方法/saveOrUpdate()方法儲存到表格內，或經由Session物件的get()或load()方法由表格讀出的物件，這兩類物件將會存放在Session的快取(緩衝區)內，受到Hibernate的監控，我們稱這樣的物件為永續物件。
- 只要Session物件沒有關閉，永續物件的屬性如果有任何改變，Hibernate會在程式執行交易的commit()方法時呼叫Session介面的flush()方法更新表格中與永續物件對應的紀錄。
- 若程式執行Session物件的delete()/remove()方法刪除某個永續物件，Hibernate也會刪除表格中與之對應的紀錄。此時永續物件由於失去了表格中對應的紀錄，會進入移除(Removed)狀態而成為移除物件。

# Detached(分離)狀態

---

- 永續物件會於程式執行Session介面的close()方法/evict(obj)方法/clear()方法後轉為分離狀態，其OID與表格某筆紀錄的主鍵值對應，但不存在於Session的緩衝區內，所以不受Hibernate監控。
  - 由get()方法/load()方法得到的物件，若程式關閉(close)Session物件，該物件會由永續狀態進入分離狀態而不受Hibernate的監控。程式修改分離物件的屬性值不會對與之對應紀錄內的欄位造成任何影響。
- 分離物件可經由Session物件的merge()/update()/saveOrUpdate()方法更新表格中與之對應的紀錄，並放入Session的緩衝區，此時該物件由分離狀態轉變為永續狀態。



# Removed(移除)狀態

- 如果程式使用Session物件的delete()或remove()方法移除了永續物件，永續物件便進入移除狀態。與之對應的紀錄稍後將會被Hibernate刪除。
- 程式不應該繼續使用一個成為移除狀態的物件，應該釋放任何參考它的變數，讓該物件在適當的時候被Garbage Collection回收。

狀態	表格是否存在 對應的紀錄	是否位於 Session緩衝區	是否含有 Object Identifier
Transient(臨時)	N	N	N
Persistent(永續)	Y	Y	Y
Detached(分離)	Y	N	Y
Removed(移除)	N	N	Once owned

# JUnit 單元測試工具

- 在JAVA的環境中領域裡，最常使用的單元測試工具。
- 需在專案的pom.xml內加入下列dependency才能使用JUnit:  
<dependency>  
    <groupId>junit</groupId>  
    <artifactId>junit</artifactId>  
    <version>4.12</version>  
    <scope>test</scope>  
</dependency>
  - 4.0以後的版本才能使用在測試程式中使用註釋
- 新建JUnit單元測試的Java程式
  - 在src/test/java下新建套件: jut
  - 點選jut，右按>new>other>在上面的文字方塊輸入JUnit，點選JUnit Test Case>Next
  - 於Name欄位輸入程式名稱: MyJUnitTest
  - 勾選兩個選項: setUp與tearDown，按下Finish，完成。

# JUnit 單元測試工具

..

- 程式中有三個方法分別以@Before，@After 與@Test標示。以@Test標示的方法為測試方法。
- 要執行測試方法的步驟：
  - 點選測試方法名稱>右按>Run As>JUnit Test
- 當執行測試方法時，@Before標示的方法會最先執行，然後執行@Test標示的方法，最後執行@After標示的方法。

@Before

```
public void init() {  
    factory = HibernateUtils.getSessionFactory();  
    session = factory.openSession();  
    tx = session.beginTransaction();  
}
```

@After

```
public void destroy() {  
    try {  
        System.out.println("3.-----");  
        tx.commit();  
        System.out.println("4.-----");  
        session.close();  
        factory.close();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

# Session介面提供的方法

- `session.save(obj)`: 儲存一個臨時物件
- `session.persist(obj)`: 儲存一個臨時物件
- `session.get(Class, id)`: 讀取一個永續物件
- `session.load(Class, id)`: 讀取一個永續物件
- `session.flush()`: 讓session內的物件與表格中對應的紀錄同步
- `session.delete(obj)`: 刪除一個分離或永續物件
- `session.update(obj)`: 更新一個永續物件
- `session.merge(obj)`: 更新一個分離物件
- `session.saveOrUpdate(obj)`: 新增或更新一個分離物件
- `session.clear()`: 清除所有存放在Session緩衝區內的物件
- `session.evict(obj)`: 清除存放在Session緩衝區內的obj物件
- `session.close()`: 關閉Session物件
- `session.createQuery(hql)`: 依照條件讀取一組永續類別的物件

範例程式為\src\test\java之下的  
`ch03._00.MethodsOfSession.java`

# Session介面的save()

---

- 語法: Serializable save(Object obj)
- 方法會將臨時物件(參數 obj)儲存到表格內，參數obj成為永續物件。如果程式採用資料庫提供的主鍵自增機制，則Hibernate會立即發送SQL敘述，將物件寫入表格。如果程式自行提供主鍵值，Hibernate不會立即將物件寫入表格，等到執行交易的commit()方法後才會將物件實際寫入資料庫。
- 傳回值: 物件的鍵值
- 如果是採用資料庫底層自增的機制產生oid，不需要替臨時物件準備oid

# Session介面的save()

// 使用Session介面的save()儲存臨時物件，如果紀錄的主鍵值是使用表格自增欄位的方式產生，  
// save()會立即發送SQL敘述，此時臨時物件不需要提供OID，因為無論OID為null或是任何整數，  
// 都會被資料庫產生的鍵值覆蓋。

@Test

```
public void saveDemo01() {  
    // 新建三個BookBean物件，目前為臨時物件  
    BookBean bb1 = new BookBean(1000, "美麗人生", "孫玲", 450.0, 150);  
    BookBean bb2 = new BookBean(null, "健康人生", "張安國", 350.0, 100);  
    BookBean bb3 = new BookBean(null, "幸福人生", "楊磊", 380.0, 325);  
    // 利用session.save()儲存臨時物件，儲存後為永續物件  
    session.save(bb1); // 雖然save()方法對bb1物件儲存多次，但只會寫入表格一次  
    session.save(bb1);  
    session.save(bb1);  
    session.save(bb2);  
    session.save(bb3);  
    bb1.setPrice(950.0); // 永續物件位於Session快取中，受Hibernate監控，修改永續物件不需要  
    bb2.setPrice(960.0); // 執行Session介面的update()方法  
    bb3.setPrice(970.0);  
}
```

# Session介面的save()

..

// 修改永續物件的鍵值，會在執行tx.commit()時丟出例外。

@Test

```
public void saveDemo02() {  
    BookBean bb1 = new BookBean(1000, "美麗人生-續集", "孫玲", 450.0, 100);  
    BookBean bb2 = new BookBean(null, "健康人生-續集", "張安國", 350.0, 80);  
    BookBean bb3 = new BookBean(null, "幸福人生-續集", "楊磊", 380.0, 75);  
    Serializable o = session.save(bb1); // 儲存後bb1為永續物件  
    System.out.println("Key=" + o);  
    o = session.save(bb2); // 儲存後bb2為永續物件  
    System.out.println("Key=" + o);  
    o = session.save(bb3); // 儲存後bb3為永續物件  
    System.out.println("Key=" + o);  
    // 下列敘述會丟出例外  
    // bb1.setBookId(1000);  
    System.out.println("永續物件不能改鍵值，否則會在執行tx.commit()時丟出例外");  
}
```

# Session 介面的 save()

...

```
// 程式自行提供主鍵值
@Test
public void saveDemo03() {
    Animal a1 = new Animal(10, "史努比");
    Animal a2 = new Animal(20, "貓凱蒂");

    Serializable o = session.save(a1); // 儲存後a1為永續物件
    System.out.println("Key=" + o);
    o = session.save(a2);             // 儲存後a2為永續物件
    System.out.println("Key=" + o);
}
```



# Session介面的persist()方法

- 語法：void persist(Object obj)
- 將一個處於臨時狀態的物件(參數 obj)儲存到表格內，參數obj成為永續物件。此方法必須在交易內執行。
- 如果程式採用資料庫提供的主鍵自增機制，以persist()儲存的臨時物件絕對不能有OID，否則會得到下列錯誤訊息：  
org.hibernate.PersistentObjectException: detached entity passed to persist
- 如果程式採用自行提供主鍵值，以persist()儲存的臨時物件絕對要有OID
- save()儲存的臨時物件可以有OID，此OID會被資料庫自增的鍵值覆蓋。
- save()的傳回值型態為java.io.Serializable，而persist()沒有傳回值。

# Session介面的persist()

---

@Test

```
public void persistDemo01() {
```

```
    BookBean bb1 = new BookBean(null, "永續經營自己的人生", "張芳芳", 450.0, 60);
```

```
    BookBean bb2 = new BookBean(null, "吃出健康，活的快樂", "張芳芳", 360.0, 72);
```

```
    session.persist(bb1);
```

```
    session.persist(bb2);
```

```
}
```

@Test

// Session#persist() 儲存的臨時物件如果有OID，會得到

// org.hibernate.PersistentObjectException: detached entity passed to persist:

// ch03.\_00.model.BookBean

```
public void persistDemo02() {
```

```
    try {
```

```
        // 下面的敘述因為有OID而丟出例外
```

```
        BookBean bb = new BookBean(1001, "正確規劃自己的人生", "劉啟民", 510.0, 48);
```

```
        session.persist(bb);
```

```
    } catch (Exception e) {
```

```
        e.printStackTrace();
```

```
    }
```

```
}
```

# Session介面的persist()

---

@Test

```
public void persistDemo03() {  
    Animal a1 = new Animal(30, "嘉菲貓");  
    Animal a2 = new Animal(40, "高飛狗");  
    session.persist(a1); // 儲存後a1為永續物件  
    session.persist(a2); // 儲存後a2為永續物件  
}
```

# Session介面的get()與load()方法

- get()與load()兩方法都可依主鍵讀取表格內的資料，但作法不同。
- 它們的用法如下：  
Member member = session.**get**(Member.class, id); // id: 主鍵值  
Member member = session.**load**(Member.class, id); // id: 主鍵值  
- 第一個參數表示要讀取之物件所屬類別，第二個參數要讀取物件的主鍵值
- load()方法用在讀取的永續物件是真實存在，此方法並未真正讀取表格內的紀錄，它傳回一個沒有任何內容的代理物件(稱之為**proxy**)。實務上很少使用此方法
- 當後面的程式碼需要使用代理物件內的屬性值時，如執行了永續物件的getXXX()或toString()方法時，Hibernate才會送出相關的SQL敘述到資料庫去讀取相關的紀錄。
- load()方法展示Hibernate發明的一種可增進程式執行效能的技術：Lazy Loading。
- 讀取時如果發現該鍵值對應的紀錄並不存在，程式會丟出ObjectNotFoundException例外

# Session介面的get()與load()方法 ..

- get()方法應該用在不確定要讀取的永續物件是否存在，執行此方法時Hibernate會立刻送出對應的SQL敘述去資料庫中讀取表格資料，並傳回要讀取的永續物件。若與鍵值對應的紀錄不存在，程式會傳回null。
- 依鍵值讀取表格內的紀錄應該使用get()而不該使用load()方法。
- 在讀取表格內的資料時，session物件不能關閉，否則程式與資料庫的連線不復存在而無法讀取資料，程式會丟出如下的例外：  
`org.hibernate.LazyInitializationException: could not initialize proxy  
[ch03._00.model.BookBean#2] - no Session`
- 在Hibernate中以get方法()讀取物件時，該物件若含有多方屬性，Hibernate預設採用Lazy Loading的技術處理多方屬性的內含值，亦即不會立刻讀取多方表格中對應的紀錄。
  - OneToMany、ManyToOne的多方預設採用Lazy Loading的技術處理多方屬性的內含值

# Session介面的get()

---

// get()方法會去讀取資料庫，傳回要讀取的永續物件。如果永續物件不存在，程式會傳回null。

@Test

```
public void getDemo01() {  
    BookBean bb1 = session.get(BookBean.class, 2); // 將鍵值改為20再試一次  
    if (bb1 != null) {  
        System.out.println(bb1.getAuthor() + ", " + bb1.getPrice());  
    } else {  
        System.out.println("找不到bb物件");  
    }  
}
```

# Session介面的load()

// load()方法會傳回一個代理物件，而沒有真正去讀取資料庫內的資料。

// 當程式需要使用物件的屬性值時，Hibernate才會進行資料存取。若果發現該永續物件不存在，

// 程式會丟出例外。

@Test

```
public void loadDemo01() {  
    BookBean bb1 = session.load(BookBean.class, 3);  
    System.out.println("load()執行完畢...");  
    System.out.println("-----");  
    System.out.println("代理類別的全名: " + bb1.getClass().getName());  
    System.out.println("-----");  
    System.out.println("-----準備讀取BookBean物件的內容-----");  
    // session.close(); // 如果執行session.close()關閉session，程式會發生Exception:  
    // could not initialize proxy [_00.model.BookBean#6] - no Session  
    System.out.println(bb1.getAuthor() + ", " + bb1.getPrice());  
}
```

# Session介面的flush()方法

---

- 語法

`void flush()`

- 要求Hibernate將Session快取中的物件與資料庫進行同步，即有任何永續物件的屬性值與對應之紀錄欄位內容不一致，本方法會發出UPDATE SQL敘述以更新對應的紀錄。
- 預設情況當程式執行HQL進行查詢與commit交易前，Hibernate會自動執行session.flush()方法。
- flush()可能會發送UPDATE的SQL敘述，也可能不會發送。視是否有永續物件的屬性值與對應之紀錄欄位內容不一致而定。
- 若有需要我們的程式也可主動執行Session的flush()方法。



# Session介面的flush()

// 主動呼叫session.flush()方法，如果有任何永續物件的屬性值與表格內的紀錄欄位不一致

// 本方法會發出UPDATE SQL敘述

@Test

```
public void flushDemo01() {  
    BookBean bb1 = session.get(BookBean.class, 3);  
    if (bb1 == null) {  
        System.out.println("該物件不存在");  
        return;  
    }  
    bb1.setAuthor("劉麗芳-3");  
    bb1.setStock(50);  
    System.out.println("1.-----"),  
    session.flush();  
    System.out.println("2.-----");  
}
```

// 還要搭配觀察

@After

```
public void destroy() {
```

```
    // ...
```

```
}
```

送出的訊息

# Session介面的flush()

---

// 主動呼叫session.flush()方法，如果沒有任何永續物件的屬性值被修改

// 本方法不會發出UPDATE SQL敘述

@Test

```
public void flushDemo02() {  
    BookBean bb1 = session.get(BookBean.class, 2);  
    if (bb1 == null) {  
        System.out.println("該物件不存在，請修改鍵值重新執行");  
        return;  
    }  
    bb1.setStock(125);  
    bb1.setStock(150);  
    bb1.setStock(200);  
    System.out.println("1.-----");  
    session.flush();  
    System.out.println("2.-----");  
}
```

# Session介面的flush()

// 主動呼叫session.flush()方法，如果沒有任何永續物件的屬性值被修改

// 本方法不會發出UPDATE SQL敘述

@Test

```
public void flushDemo03() {  
    BookBean bb1 = session.get(BookBean.class, 2);  
    if (bb1 == null) {  
        System.out.println("該物件不存在，請修改鍵值重新執行");  
        return;  
    }  
    int stock = bb1.getStock();  
    bb1.setStock(50);  
    bb1.setStock(150);  
    bb1.setStock(250);  
    bb1.setStock(stock);  
    System.out.println("1.-----");  
    session.flush();  
    System.out.println("2.-----");  
}
```

// 還要搭配觀察 destroy()送出的訊息

# Session介面的delete()方法

---

- 語法

`void delete(Object obj)`

- 可依參數obj刪除一個分離或永續物件對應的紀錄。
  - 若參數物件的OID與某筆紀錄的主鍵相同就刪除該紀錄。
  - 如參數物件的OID沒有對應某筆紀錄則丟出例外。
- `session.delete(obj)`方法只能刪除永續物件或分離物件，刪除臨時物件無任何"刪除"意義可言。
- `session.delete(obj)`: 若用以刪除臨時物件會得到錯誤訊息：

[Batch update returned unexpected row count from update [0]; actual row count: 0; expected: 1]

# Session介面的delete()

// 1. session.delete(obj)方法只能刪除永續物件與分離物件，刪除臨時物件無任何"刪除"意義可言。

// 2. session.delete(obj): 若用以刪除臨時物件 會得到錯誤訊息[Batch update returned

// unexpected row count from update [0]; actual row count: 0; expected: 1]

@Test

```
public void deleteDemo01() {
```

```
    // 執行session.get()後，bb0 永續物件
```

```
    BookBean bb0 = session.get(BookBean.class, 1);
```

```
    session.delete(bb0);          // 刪除永續物件
```

```
    //
```

```
    BookBean bb1 = session.get(BookBean.class, 2);
```

```
    tx.commit();
```

```
    session.close();
```

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

```
    tx = session.beginTransaction();
```

```
    session.delete(bb1); // 刪除分離物件
```

```
    System.out.println("此時bb0, bb1物件處於Removed狀態");
```

```
}
```

# Session介面的delete()

@Test // 使用 session.delete() 刪除臨時物件會在執行tx.commit()時丟出例外。

```
public void deleteDemo02() {
```

```
    // 新建BookBean物件後，bb 為臨時物件，下面的敘述刪除臨時物件
```

```
    BookBean bb = new BookBean(101, "校長流浪記-9", "張梅芳-9", 350.0, 600);
```

```
    session.delete(bb); // 用session.delete()方法刪除臨時物件會丟出例外。
```

```
}
```

@Test // 展示刪除分離物件。

```
public void deleteDemo03() {
```

```
    BookBean bb = session.get(BookBean.class, 3);
```

```
    if (bb != null) {
```

```
        tx.commit();
```

```
        session.close(); // 關閉Session後，bb分離物件
```

```
        System.out.println("關閉Session後bb物件處於Detached狀態");
```

```
        session = factory.openSession(); // 重新開啟交易
```

```
        tx = session.beginTransaction();
```

```
        session.delete(bb); // 此時 bb 為分離物件
```

```
        System.out.println("刪除bb物件後，它處於Removed狀態");
```

```
    } else {
```

```
        System.out.println("查無此筆紀錄，請修改鍵值重新執行");
```

```
    }
```

```
}
```

# Session介面的delete()

---

@Test

```
public void deleteDemo04() {  
    BookBean bb2 = new BookBean();  
    bb2.setBookId(8);  
    System.out.println("嘗試刪除分離物件bb2:, 主鍵為8, 只有主鍵值而無其它屬性");  
    System.out.println("如果該物件存在, 可刪除, 如果該物件不存在會丟出例外");  
    session.delete(bb2);  
}
```

# Session介面的update()方法

---

- 語法  
void update(Object obj)
- 更新一個分離物件(參數 obj)，更新後參數obj會成為永續物件。此方法必須在交易(Transaction)內執行，程式先執行session物件的beginTransaction()開啟一個交易，然後才透過update()方法更新分離物件。
- 執行update()方法後，Hibernate不會立即將物件寫入表格而會等到執行交易的commit()方法後才會將物件實際寫入表格。
- 傳回值：
  - 無
- 使用update()方法更新臨時物件會在執行tx.commit()時丟出例外。



# Session介面的update()

// Session#update() 用以更新分離物件。

```
public void updateDemo01() {  
    try {  
        BookBean bb = new BookBean(null, "七天學會游泳", "林書豪", 520.0, 37);  
        session.save(bb); // 執行session.save(bb)後，bb為永續物件  
        tx.commit();  
        session.close(); // 關閉session後，bb 變為分離物件  
        session = factory.openSession(); // 開啟新session，bb不存在於這個session內  
        tx = session.beginTransaction();  
        bb.setTitle("八天學會游泳"); // 修改bb的title性質  
        bb.setAuthor("林書濠"); // 修改bb的作者性質  
        session.update(bb); // session.update(bb)更新分離物件。  
    } catch (Exception e) {  
        e.printStackTrace();  
        tx.rollback();  
    }  
}
```

# Session介面的saveOrUpdate()方法

---

- 語法

`void saveOrUpdate(Object obj)`

- 此方法會對參數obj進行save(obj)或update(obj)，依據的準則為obj的鍵值是否已經存在於資料庫表格內。
- 如果參數obj處於分離狀態，則此方法會自動呼叫update(obj)更新對應的表格紀錄，執行SQL UPDATE敘述
- 如果參數obj處於臨時狀態，則此方法會自動呼叫save(obj)將obj新增到表格內，執行SQL INSERT INTO敘述
- 傳回值：
  - － 無

# Session介面的saveOrUpdate()

@Test

```
public void saveOrUpdateDemo01() {  
    BookBean bb1 = session.get(BookBean.class, 8);  
    if (bb1 == null) {  
        System.out.println("BookBean物件不存在，請調整主鍵值");  
        return;  
    }  
    tx.commit();  
    session.close();  
    session = factory.openSession();  
    tx = session.beginTransaction();  
    // 此時bb1為分離(detached)物件  
    bb1.setAuthor("劉麗芳-5");  
    bb1.setStock(65);  
    BookBean bb2 = new BookBean(null, "當Spring遇見Summer", "李建中", 330.0, 18);  
    // 此時，目前的session物件中沒有控管任何永續物件。  
    // 但程式中有一個分離(detached): bb1, 另外有一個臨時(transient)物件: bb2  
    System.out.println("=====");  
    System.out.println("1.--準備執行saveOrUpdate(bb1), bb1為分離(detached)物件---");  
    session.saveOrUpdate(bb1);  
    session.flush();  
    System.out.println("2.--準備執行saveOrUpdate(bb2), bb2為臨時(transient)物件---");  
    session.saveOrUpdate(bb2);  
    session.flush();  
}
```

# Session介面的merge()方法

---

merge()的使用方式如下：

- 新建(new)一個物件bb1，沒有OID，執行session.merge(bb1)，會發出SQL INSERT INTO敘述，此時session.merge(bb1)與session.save(bb1)效果一樣。
- 新建(new)一個物件bb2，有OID，而且表格內有紀錄與之對應(即存在一筆記錄，其主鍵值與OID相等)，即此物件為分離物件，執行session.merge(bb2);時會發SQL UPDATE敘述。
- 新建(new)一個物件bb3，有OID，但表格內沒有紀錄與之對應(即沒有任何一筆記錄其主鍵值與OID相等)，即此物件為transient物件，執行session.merge(bb3);時，會發SQL INSERT INTO敘述。
- 執行**merge()**方法後，傳入此方法的參數不會成為永續物件，但此方法的傳回值是永續物件。

# Session介面的merge()

---

// 展示merge()處理沒有OID的臨時物件

@Test

public void mergeDemo01() {

// case1: 新建一個物件bb1，沒有OID，故Session快取內不可能有該物件

BookBean bb1 = new BookBean(null, "當薪水調高時-3", "李建國-3", 410.0, 33);

System.out.println("case 1.--準備執行merge(臨時物件)---");

System.out.println("新建一個物件bb1，沒有OID.，執行session.merge(bb1)，"  
+ "會發出SQL INSERT INTO敘述");

// 效果與session.save(bb1)一樣

session.merge(bb1);

}

# Session介面的merge()

---

// 展示merge()處理有OID的分離物件

@Test

public void mergeDemo02() {

// case2: 新建一個物件bb2(即Session快取內沒有該物件)，有OID, 而且表格內有紀錄與之對應

BookBean bb2 = new BookBean(5, "當Sally遇見Hally第4集", "李建國", 540.0, 14);

System.out.println("case 2.--準備執行merge(bb2)---");

// OK, 會發出UPDATE的SQL敘述更新表格中主鍵等於OID的紀錄

BookBean bb = (BookBean) session.merge(bb2);

}

# Session介面的merge()

---

@Test

```
public void mergeDemo03() {
```

```
    // case3: 新建一個物件bb3(即Session快取內沒有該物件)，有OID, 但表格內沒有紀錄與之對應
    BookBean bb3 = new BookBean(301, "當Sally遇見Hally-3", "李建國-303", 410.0, 303);
    session.merge(bb3); // OK, 會發出SQL INSERT INTO敘述，忽略原有的OID
    session.save(bb3);  // OK, 與session.merge()完全相同，即會發出SQL INSERT INTO敘述
                        //，忽略原有的OID
    // session.update(bb3); // NG, Row was updated or deleted by another transaction
    // (or unsaved-value mapping was incorrect) :
    // session.saveOrUpdate(bb3); // NG, Row was updated or deleted by another
    // transaction (or unsaved-value mapping was incorrect) :
}
```

# Session介面的merge()

```
/*
session.merge(obj): 本方法會傳回更新或新增後的物件，此物件為永續物件。
傳入的參數不會成為永續物件，參數可能是：
    1. 分離物件(具有有效的主鍵值，更新它)
    2. 臨時物件(OID為 null，或沒有對應的主鍵值，新增它)
*/
@Test
public void mergeDemo04() {
    BookBean bb1 = new BookBean(12, "快樂人生第二集", "張美春", 380.0, 25);
    BookBean bbR1 = (BookBean) session.merge(bb1);
    BookBean bb2 = new BookBean(null, "快樂人生第三集", "張大春", 390.0, 27);
    BookBean bbR2 = (BookBean) session.merge(bb2);
    BookBean bb3 = new BookBean(1000, "快樂人生第四集", "張小春", 400.0, 20);
    BookBean bbR3 = (BookBean) session.merge(bb3);

    System.out.println("-----");
    bb1.setAuthor("張春芳-111(參數)");
    bb2.setAuthor("張春芳-222(參數)");
    bb3.setAuthor("張春芳-333(參數)");
    // bbR1.setAuthor("張春芳-111(傳回值)");
    // bbR2.setAuthor("張春芳-222(傳回值)");
    // bbR3.setAuthor("張春芳-333(傳回值)");
}
```



# Session介面的clear()/evict()/close()

---

- clear()

- 清除存放在Session緩衝區內的所有永續物件。處於Session緩衝區內，受到Hibernate監控的所有物件都不再與該Session發生關聯，也就是原本的永續物件都會變為分離物件。
- 呼叫此方法時，Hibernate不會對資料庫發出任何SQL敘述。
- 程式必須開啟交易才能呼叫此方法。

- evict(obj)

- 只清除Session緩衝區內的obj物件，即obj物件不再受到Hibernate監控，當Session結束時不會對obj物件的異動發出SQL敘述。

- close()

- session.close()方法關閉Session，意味著結束session並釋放JDBC連線。

# Session介面的clear()

---

@Test

```
public void clear01() {  
    BookBean bb1 = session.get(BookBean.class, 15);  
    System.out.println("bb1的作者: " + bb1.getAuthor());  
    bb1.setAuthor("丁丁");  
    session.clear();  
    System.out.println("-----");  
    bb1 = session.get(BookBean.class, 15);  
    System.out.println("bb1的作者: " + bb1.getAuthor());  
    System.out.println("觀察程式是否發出兩次SELECT敘述..., ");  
    System.out.println("然後註解session.clear();後再執行一次，理解其中的差異");  
}
```

# Session介面的clear()

---

@Test

```
public void clear02() {  
    BookBean bb1 = session.get(BookBean.class, 5);  
    BookBean bb2 = session.get(BookBean.class, 6);  
    System.out.println("bb1的作者: " + bb1.getAuthor());  
    System.out.println("bb2的作者: " + bb2.getAuthor());  
    bb1.setAuthor("黃中和");  
    bb2.setAuthor("李良華");  
    session.clear();  
    System.out.println("觀察表格內的紀錄是否有變動..., ");  
    System.out.println("然後註解session.clear();後再執行一次，理解其中的差異");  
}
```

# Session介面的evict()

---

@Test

```
public void evict() {  
    BookBean bb1 = session.get(BookBean.class, 11);  
    BookBean bb2 = session.get(BookBean.class, 12);  
    BookBean bb3 = session.get(BookBean.class, 13);  
    bb1.setAuthor("孫悟空");  
    bb2.setAuthor("朱八戒");  
    bb3.setAuthor("沙悟淨");  
    session.evict(bb1);  
    session.evict(bb3);  
    System.out.println("觀察表格內的紀錄是否有變動...");  
}
```

# Session介面的refresh()

---

@Test

```
public void refresh() {  
    BookBean bb1 = session.get(BookBean.class, 15);  
    System.out.println("bb1的作者: " + bb1.getAuthor());  
    System.out.println("=====按下任意鍵繼續=====...");  
    try {  
        System.in.read();  
    } catch (Exception e) {  
    }  
    session.refresh(bb1);  
    System.out.println("bb1的作者: " + bb1.getAuthor());  
}
```

# 用 openSession() 方法得到 Session 物件

---

```
Session session = HibernateUtils.getSessionFactory().openSession();  
Transaction tx = null;  
try {  
    tx = session.beginTransaction();  
    //在此執行session物件的save(), update(), delete(), merge()等方法  
    tx.commit();  
} catch(Exception e){  
    if (tx!=null) tx.rollback();  
    throw new RuntimeException(e); // 由控制器捕捉此例外  
} finally {  
    if (session != null){  
        session.close();  
    }  
}
```

由 openSession() 得到的 Session 物件程式一定要執行 session.close();

# 用 `getCurrentSession()` 方法得到 Session 物件

```
Session session = HibernateUtils.getSessionFactory().getCurrentSession();
```

```
Transaction tx = null;
```

```
try {
```

```
    tx = session.beginTransaction();
```

```
    //在此執行session物件的save(), update(), delete(), merge()等方法
```

```
    tx.commit();
```

```
    memberIDList.add(mem.getUserId());
```

```
} catch (Exception e) {
```

```
    if (tx != null) tx.rollback();
```

```
    throw new RuntimeException(e);
```

```
}
```

- 由 `getCurrentSession()` 得到的 Session 物件不需要執行 `close()`。當程式執行 `tx.commit()` 或 `tx.rollback()` 時就會自動關閉 Session 物件。
- 要使用 `SessionFactory#getCurrentSession()` 取得 Session 物件，一定要在組態檔內編寫如下的標籤：

```
<property name="hibernate.current_session_context_class">thread</property>
```

否則會得到這樣的錯誤訊息：

**org.hibernate.HibernateException: No CurrentSessionContext configured!**

# 練習三

---

- 修改WebAppLab01專案的Dao



---

## 四、延遲加載

### Lazy Loading

# 延遲加載Lazy Initialization (aka Lazy Loading)

---

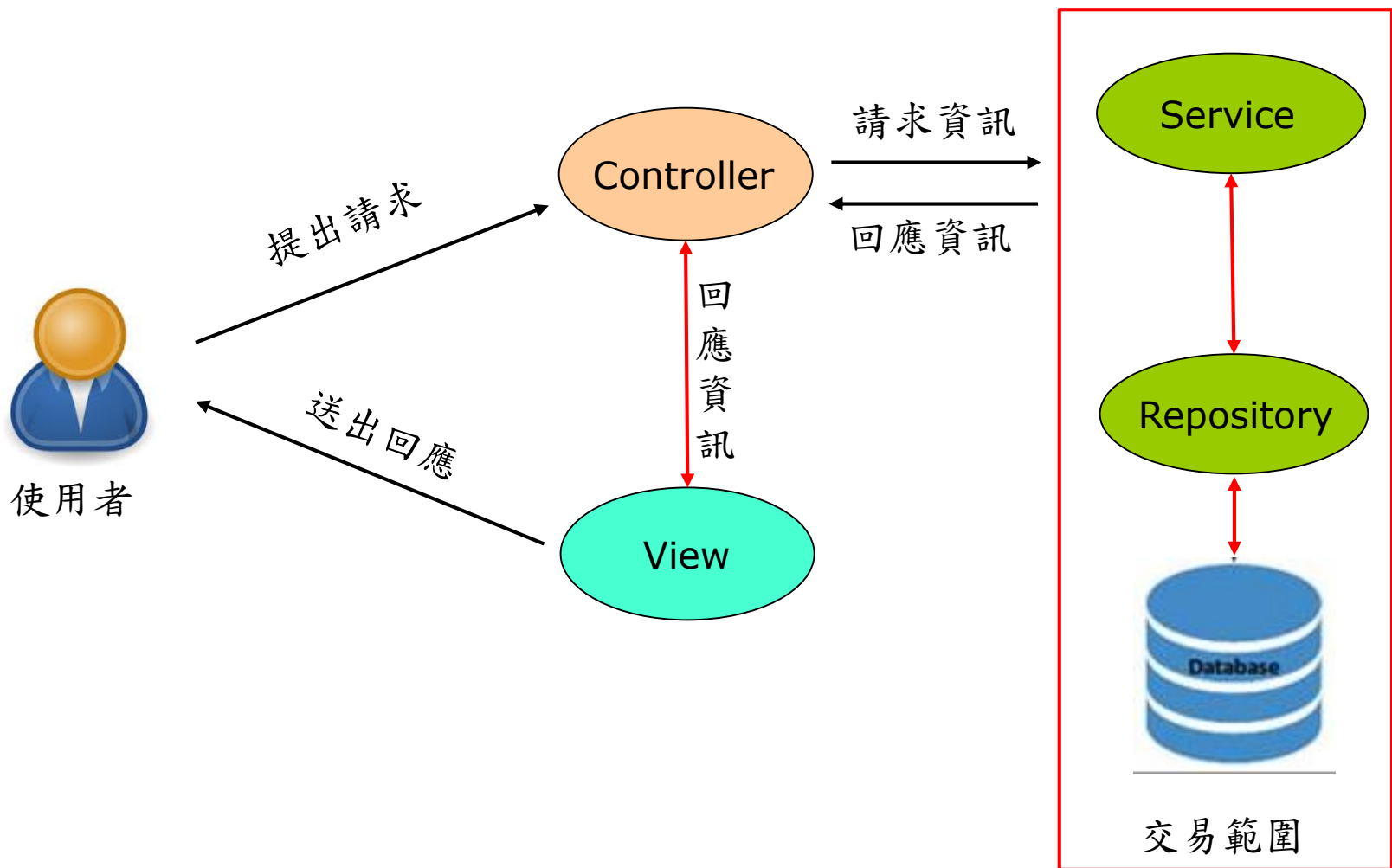
- 當程式使用`session.load()`方法載入某個物件或讀取之物件含有One-Many/Many-Many的多方成員時，Hibernate不會立即發出SELECT敘述讀取物件或多方成員而會傳回一個或多個代理物件(**proxy**)，其內的屬性值都是null。稍後程式需要使用代理物件的屬性值時，Hibernate才會發出SQL SELECT敘述讀取對應的紀錄以便組合為一個物件或多方物件。
- 在JSP程式中透過EL取出代理物件的屬性時，如果Session已經關閉則無法與資料庫保持連線，程式發出的SELECT敘述不能送至資料庫伺服器，就會發生下列的例外：

`org.hibernate.LazyInitializationException: could not initialize proxy - no Session`

# 在MVC架構下Http請求的處理流程

- Service元件開啟交易，編寫try-catch-finally區塊。在try區塊中依序呼叫所有Repository(DAO)元件的方法然後執行交易的commit()；catch區塊負責捕捉Repository元件丟出的例外然後執行交易的rollback()。
  - Service元件與Repository元件將使用Session工廠的getCurrentSession()取得Session物件
- 若所有Repository元件的方法都正常結束，回到Service元件後執行交易的commit()，若有任何Repository元件的方法拋出例外，由Service元件的catch區塊攔截例外後執行交易的rollback()。
- Service元件完成工作後回傳(Repository元件取出的)資料給控制器，控制器將得到的資料放入請求物件內，使其成為屬性物件，然後轉發(forward)給JSP，由JSP使用EL取出屬性物件內的資料以便產生完整的HTML文件。
- 此時若屬性物件含有Hibernate準備的代理物件，程式將因Session已經關閉而無法連結資料庫，程式將會丟出類似下列的錯誤訊息：  
org.hibernate.LazyInitializationException: could not initialize proxy - **no Session**

# Http請求的處理流程



# 解決Lazy Loading的做法

---

1. 透過註釋說明該類別的集合成員不採用延遲加載的方式，亦即要立即載入，例如

```
@OneToMany( mappedBy = "category", fetch = FetchType.EAGER )  
private Set<ProductEntity> products;
```

@OneToMany與@ManyToOne預設會使用延遲加載

```
@OneToMany( mappedBy = "category", fetch = FetchType.LAZY )  
private Set<ProductEntity> products;
```

# 解決Lazy Loading的做法

..

## 2. 修改DAO類別的寫法

- 利用Hibernate類別的靜態方法initialize(), 強迫Hibernate讀取實際資料庫內的資料填入Proxy物件。
- 例如：

```
try {  
    tx = session.beginTransaction();  
    Integer iid = Integer.valueOf(id);  
    member = (Member)session.load(Member.class, iid);  
    Hibernate.initialize(member);  
    tx.commit();  
} catch (Exception e){  
    if (tx != null) tx.rollback();  
    throw new RuntimeException(e);  
} finally {  
    //.....  
}
```

# 解決Lazy Loading的做法

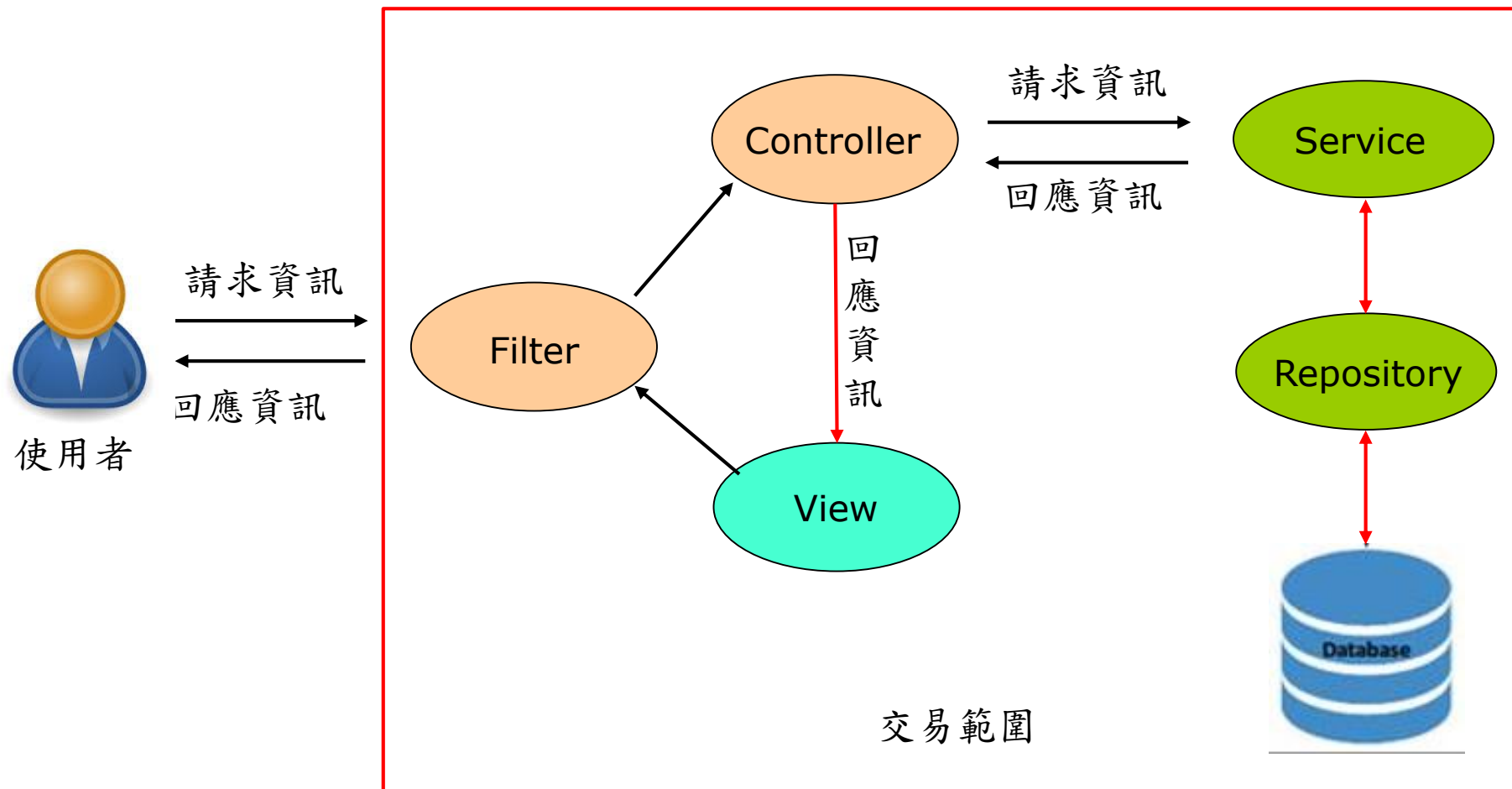
...

3. 對於Web Application，可以透過Filter延後關閉Session的時間：

- 新建一個Filter，在它的doFilter()方法內呼叫SessionFactory的getCurrentSession()取出Session物件、接著啟動交易(tx, Begin Transaction)，然後呼叫chain.doFilter(req, resp)，由被監控的資源開始執行其應有的工作(例如多個表格資料的查詢等)，等到該資源執行完所有的資料庫存取後再度回到Filter，如果一切正常，執行交易(tx)的commit()，若執行被監控的資源時拋出任何例外，則執行交易(tx)的rollback()。
- 修改Service類別。將所有呼叫Transaction之commit()/rollback()的敘述都移除，不要在**Service**類別內關閉**Session**。

注意：以getCurrentSession()方法得到Session物件會在Transaction commit()或 rollback()時自動關閉。

# 以Filter解決延遲載入可能發生的問題





# 解決Lazy Loading的做法

....

4. Spring框架為企業級應用系統提供了廣泛、完善的基礎架構，以減輕程式設計師的負擔，讓他們專心編寫企業邏輯。對於多數應用系統都會面臨的延遲加載問題，Spring框架也提供解決方案。

Spring框架替網路應用系統編寫可解決延遲加載的Filter，網路應用系統只需要引入OpenSessionInViewFilter就可用Filter解決延遲加載的問題。

在web.xml檔內定義OpenSessionInViewFilter

```
<!-- OpenSessionInViewFilter 此做法只適用於Hibernate 4 -->
<filter>
  <filter-name>HibernateFilter</filter-name>
  <filter-class>
    org.springframework.orm.hibernate4.support.OpenSessionInViewFilter
  </filter-class>
  <init-param>
    <param-name>sessionFactoryBeanName</param-name>
    <!-- 對應Spring組態檔的SessionFactory的名稱 -->
    <param-value>sessionFactory</param-value>
  </init-param>
</filter>
```

# 解決延遲加載例外的Filter

---

```
package ch04.ex03.web;
```

```
// 省略部分import敘述
```

```
@WebFilter(urlPatterns = { "/ch04/ex03/queryDepartmentById.do" })
```

```
public class HibernateFilter implements Filter {
```

```
    @SuppressWarnings("unused")
```

```
    private FilterConfig fConfig;
```

```
    private SessionFactory factory;
```

```
    public void init(FilterConfig fConfig) throws ServletException {
```

```
        this.fConfig = fConfig;
```

```
        factory = HibernateUtils.getSessionFactory();
```

```
    }
```

```
    public void destroy() {
```

```
        factory.close();
```

```
    }
```

# 解決延遲加載例外的Filter

---

```
public void doFilter(ServletRequest request, ServletResponse response,
                    FilterChain chain) throws IOException, ServletException {
    Transaction tx = null;
    try {
        Session session = factory.getCurrentSession();
        tx = session.beginTransaction();
        // 下一敘述會啟動控制器(Servlet), DAO, 視圖(JSP) ,
        // 這些程式執行時Session都保持在開啟狀態
        chain.doFilter(request, response);
        tx.commit();
    } catch (Exception e) {
        if (tx != null)
            tx.rollback();
        e.printStackTrace();
    }
}
```

# 配合的Service類別

---

```
package ch01.model.service.impl;

// 省略import敘述

public class DepartmentServiceImpl implements DepartmentService {

    DepartmentDao departmentDao;
    SessionFactory factory;

    public DepartmentServiceImpl() {
        departmentDao = new DepartmentDaoImpl();
        factory = HibernateUtils.getSessionFactory();
    }

    @Override
    public Department findById(Integer id) {
        Department dept = null;
        dept = departmentDao.findById(id);
        return dept;
    }
}

// 省略後面的敘述
```

Service類別的每個方法

1. 不啟動交易
2. 不commit/rollback交易

# 配合的Dao類別

■

```
package ch01.model.dao.impl;
// 省略部分import敘述
public class DepartmentDaoImpl implements DepartmentDao {
    SessionFactory factory;
    public DepartmentDaoImpl() {
        factory = HibernateUtils.getSessionFactory();
    }

    // 經由Session介面的get()查詢資料庫內的紀錄
    public Department findById(Integer id) {
        Department dept = null;
        Session session = factory.getCurrentSession();
        // System.out.println("session=" + session);
        dept = (Department) session.get(Department.class, id);
        return dept;
    }

    public void closeFactory() {
        factory.close();
    }

    @Override
    public Object save(Department dept) {
        Object obj = null;
        Session session = factory.getCurrentSession();
```

# 配合的Dao類別

..

```
    obj = session.save(dept);  
    return obj;  
}
```

```
@Override  
public List<Department> findAll() {  
    Session session = factory.getCurrentSession();  
    List<Department> allDepartments =  
        session.createQuery("FROM ch01_Department", Department.class)  
            .getResultList();  
  
    return allDepartments;  
}
```

```
@Override  
public Department findByName(String deptName) {  
    Department dept = null;  
    Session session = factory.getCurrentSession();  
    String hql = "FROM ch01_Department WHERE depName = :dname";  
    List<Department> list = session.createQuery(hql, Department.class)  
        .setParameter("dname", deptName)  
        .getResultList();  
  
    if (!list.isEmpty()) {  
        dept = list.get(0);  
    }  
    return dept;  
}
```

---

## 五、Hibernate Association

# Hibernate的關聯

---

- 現實世界的Entity經常與其他Entity具有某種『關聯』，表示『關聯』的做法可以是直接將其他Entity的物件參考儲存在本Entity的實例變數內，或在Entity內設計一個List或Set型態的實例變數儲存與本Entity有『關聯』的所有Entity。
- Uni-Directional Association
  - 在具有關聯的一對Entity中，只有一個Entity儲存另一個Entity的物件參考，稱為單向關聯，意即只能由一方看到(找到)另外一方。
- Bi-Directional Association
  - 如果雙方都存有對方的物件參考，稱為雙向關聯，意即彼此都能看到對方(雙方都能由己方找到對方)。
- 一對一(One-to-One)
- 一對多(One-to-Many)
- 多對一(Many-to-One)
- 多對多(Many-to-Many)



# 單向一對一

- 每個校長(Principal)只能任職一所學校(School)，每個學校只能有一位校長，校長與學校的關係為一對一。
- 由校長(Principal)找出任職的學校(School)，但無法由反向查找，此為單向一對一。
- 只要校長類別內含有該校長任職之學校的物件參考，程式就可以由校長找到任職的學校。
- 以資料庫的觀點而言就是在Principal表格內增加School表格的外鍵欄位，這樣一來，由Principal表格內的記錄就可以得到School表格內的記錄。
- 除了單向一對多的關聯關係外，加入@JoinColumn註釋之類別對應的表格有外鍵欄。表示可由此類別的物件找到對照之類別的物件(們)
- @JoinColumn(name="FK\_School\_id",  
foreignKey=@ForeignKey(name = "fkc\_pri\_sch"))
  - foreignKey=@ForeignKey(name = "fkc\_pri\_sch")：定義外鍵約束的名稱

# 單向一對一的施行步驟

步驟 1	為Principal類別、School類別加上應有的註釋：@Entity, @Table, @Id, ....
步驟 2	由於程式的需求為『由校長(Principal)找出任職的學校(School)』，因此在Principal類別內定義一個儲存School類別之物件的實例變數。
步驟 3	在此實例變數之前加上@OneToOne(cascade=CascadeType.PERSIST)。同時加上@JoinColumn(name="FK_School_id")，說明在Principal類別對應之表格的外鍵名稱為『FK_School_id』（此外鍵用以儲存School表格之主鍵）

# cascade=CascadeType.PERSIST

---

- cascade=CascadeType.PERSIST的作用：當儲存的物件含有參考之物件時，Hibernate會先寫入被參考之物件，然後寫入此物件。
- 若Principal表格含有School表格的外鍵，所以只要Principal物件含有合法的School物件參考，儲存Principal物件時，會先寫入School物件，然後才寫入Principal物件。
- 若省略 cascade=CascadeType.PERSIST時，程式必須先儲存School物件，然後儲存Principal物件，否則會得到錯誤訊息：  
ERROR: HHH000346: Error during managed flush  
[org.hibernate.TransientObjectException:  
object references an unsaved transient instance - save the transient instance before flushing: ch05.one2one.\_01.anno.model.School]

```
public static void main(String[] args) {
    // 先新建物件，Principal與School物件各兩個
    Principal p1 = new Principal();
    p1.setName("張大華-Uni");
    School s1 = new School("大華國中-Uni", "新北市板橋區");
    p1.setSchool(s1);    // 由校長可以找到學校

    Principal p2 = new Principal();
    p2.setName("李小民-Uni");
    School s2 = new School("曉明國小-Uni", "桃園市中壢區");
    p2.setSchool(s2);    // 由校長可以找到學校

    SessionFactory factory = HibernateUtils.getSessionFactory();
    Session session = factory.openSession();
    Transaction tx = session.beginTransaction();
    //    如果Principal類別的@ManyToOne註釋加入cascade=CascadeType.PERSIST
    //        session.persist(s1);    // 則儲存School物件的這兩列可以省略。
    //        session.persist(s2);
    System.out.println("-----");
    session.persist(p1);
    session.persist(p2);
    tx.commit();
    session.close();
    System.out.println("程式結束(Done...!!)");
    factory.close();
}
```

```
SessionFactory factory = HibernateUtils.getSessionFactory();
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

System.out.println("找校長(單向一對一): ");
Principal p = session.get(Principal.class, 1);
if ( p != null) {
    System.out.println(p);
} else {
    System.out.println("查詢的Principal紀錄不存在");
}
System.out.println("-----");
System.out.println("找學校(單向一對一): ");
School s = session.get(School.class, 2);
if ( s != null) {
    System.out.println(s);
} else {
    System.out.println("查詢的School紀錄不存在");
}
tx.commit();
session.close();
System.out.println("程式結束(Done...!!)");
factory.close();
```

# Principal.java

---

```
package ch05.one2one._01.anno.model;
import javax.persistence.*;
// 單向一對一，可由Principal物件找到School物件，但無法由反向查找。
// 標籤: @OneToOne, @JoinColumn
@Entity
@Table(name="ch05_oo1_Principal_Table")
public class Principal {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String name;

    @OneToOne(cascade=CascadeType.PERSIST)
    @JoinColumn(name="FK_School_id", foreignKey=@ForeignKey(name =
        "fkc_pri_sch"))
    private School school;

    public Principal() {
    }

    public Principal(Integer id, String name, School school) {
        this.id = id;
        this.name = name;
        this.school = school;
    }
    // 省略Getter/Setter
}
```

# School.java

---

```
package ch05.one2one._01.anno.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "ch05_oo1_School_Table")
public class School {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String schoolName;
    private String address;

    public School() {
        super();
    }
    // 省略剩餘敘述
```

# 雙向一對一



- 雙向一對一：校長(PrincipalBi)與學校(SchoolBi) 彼此都可以找到對方。只要這兩個永續類別都定義可存對方物件參考的變數，則雙方都可以找到對方，即：可由校長找到學校，也可由學校找到校長。
- 校長(PrincipalBi)類別

```
@JoinColumn(name="school_id", foreignKey=@ForeignKey(name = "fkc_pri_sch2"))
private SchoolBi school_p;
```
- 學校(SchoolBi)類別

```
@OneToOne(mappedBy = "school_p")
PrincipalBi principal;
```



# 雙向一對一的施行步驟

步驟 1	為永續類別加上應有的註釋(@Entity, @Table, @Id, ....)
步驟 2	在PrincipalBi類別內定義一個儲存SchoolBi類別之物件的實例變數
步驟 3	在此實例變數之前加上@OneToOne與@JoinColumn。@JoinColumn是說明在PrincipalBi類別所對應之表格中，儲存外鍵的欄位名稱
步驟 4	在SchoolBi類別內定義一個儲存PrincipalBi類別之物件的實例變數 PrincipalBi principal;
步驟 5	在此實例變數之前加上@OneToOne(mappedBy="school")。 mappedBy="school" 說明本類別(SchoolBi)對應之表格並沒有呈現關聯的外鍵，外鍵的欄位資訊位於對方類別PrincipalBi的"school" 變數上。

# 雙向一對一 O2O\_02\_Main\_Bi\_Insert.java

```
package ch05.one2one._02.anno.main;
import org.hibernate.*;
import ch05.one2one._02.anno.model.PrincipalBi;
import ch05.one2one._02.anno.model.SchoolBi;
import ch05.one2one._02.anno.util.HibernateUtils;
//雙向一對一：由校長(Principal)與學校(School)彼此都可找到對方
public class O2O_02_Main_Bi_Insert {
    public static void main(String[] args) {
        SessionFactory factory = HibernateUtils.getSessionFactory();
        Session session = factory.openSession();
        PrincipalBi p1 = new PrincipalBi(null, "劉梅芳Bi");
        SchoolBi s1 = new SchoolBi(null, "梅芳高中Bi", "台北市松山區");
        p1.setSchool(s1);
        PrincipalBi p2 = new PrincipalBi(null, "曾文山Bi");
        SchoolBi s2 = new SchoolBi(null, "文山國中Bi", "台北市文山區");
        p2.setSchool(s2);
        Transaction tx = session.beginTransaction();
        session.persist(p1);
        session.persist(p2);
        tx.commit();
        session.close();
        System.out.println("程式結束(Done...!!)");
        factory.close();
    }
}
```

```
session.save(s1);
session.save(s2);
session.save(p1);
session.save(p2);
```

# 雙向一對一 O2O\_02\_Main\_Bi\_Query.java

---

```
public static void main(String[] args) {  
    SessionFactory factory = HibernateUtils.getSessionFactory();  
    Session session = factory.openSession();  
    Transaction tx = session.beginTransaction();  
    System.out.println("查找校長:");  
    PrincipalBi p = session.get(PrincipalBi.class, 1);  
    if ( p != null) {  
        System.out.println("校長:" + p);  
    } else {  
        System.out.println("查無此筆資料");  
    }  
    System.out.println("-----");  
    System.out.println("查找學校:");  
    SchoolBi s = session.get(SchoolBi.class, 2);  
    if ( s != null) {  
        System.out.println("學校:" + s);  
    } else {  
        System.out.println("查無此筆資料");  
    }  
    tx.commit();  
    session.close();  
    System.out.println("程式結束(Done...!!)");  
    factory.close();  
}
```

# PrincipalBi.java

---

```
package ch05.one2one._02.anno.model;

import javax.persistence.*;

// 雙向一對一，可由PrincipalBi物件與SchoolBi物件都可以查找對方。
// 關注標籤：@OneToOne
@Entity
@Table(name = "ch05_oo2_Principal_Table_02")
public class PrincipalBi {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer pid;
    private String name;

    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name="FK_school_sid", foreignKey=@ForeignKey(name =
    "fk_c_pri_sch2"))
    private SchoolBi school_p;

    public PrincipalBi() {
    }

    // 省略剩餘敘述
```

# SchoolBi.java

---

```
package ch05.one2one._02.anno.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "ch05_oo2_School_Table_02")
public class SchoolBi {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer sid;
    private String schoolName;
    private String address;
    @OneToOne(mappedBy = "school_p")
    PrincipalBi principal;

    public SchoolBi() {
    }
}
```

# 單向一對多

---

- 單向一對多：由部門(Department，一方)找出該部門內所有員工(Employee，多方)，但無法由員工找到所屬之部門。
  - － 口訣：一對多，一方有個儲存多方物件參考的List/Set物件，簡稱『一方有個多』
- 只要部門(Department)內含有一個能儲存多方物件(員工類別)的List/Set物件，程式就可以由部門找到其內的所有員工。
  - － 由於是單向(Unidirectional)的一對多，所以多方(Employee)不能含有一方(Department)的物件參考，否則就會形成雙向Bidirectional)。
- 以資料庫的觀點而言就是在Employee類別(多方)對應的表格內增加一個外鍵欄位，此外鍵欄位儲存Employee物件所屬之部門物件(一方)對應之紀錄的主鍵，Hibernate就可以由部門(Department)找出其內所有員工(Employee)。

# 單向一對多的施行步驟

步驟 1	為永續類別加上應有的註釋(@Entity, @Table, @Id, ....)
步驟 2	<p>由於程式的需求為『由部門(Department)找出其內的員工(Employee)』，因此在此Department類別內定義一個儲存Employee(多方)物件參考之List/Set型態的變數。</p> <pre>private Set&lt;Employee&gt; employees = new LinkedHashSet&lt;&gt;();</pre>
步驟 3	<p>在此變數前加上</p> <pre>@OneToMany(cascade=CascadeType.ALL) @JoinColumn(name = "fk_dept_id", referencedColumnName = "dept_id")</pre> <p>"One"代表本類別(Department), "Many"代表對照類別(Employee)。</p> <p>@JoinColumn的name屬性說明多方表格的fk_dept_id欄位為外鍵欄位，對照的欄位(referencedColumnName)為一方表格的dept_id欄</p>

# 單向一對多 O2MMain01\_Uni\_Insert.java .

---

```
package ch05.one2many._00.anno.main;

import java.util.*;
import org.hibernate.*;
import ch05.one2many._00.anno.model.Department;
import ch05.one2many._00.anno.model.Employee;
import ch05.one2many._00.anno.util.HibernateUtils;

public class O2MMain01_Uni_Insert {
    public static void main(String[] args) {
        public static void main(String[] args) {
            Employee emp1 = new Employee(null, "CUS001", "黃華");
            Employee emp2 = new Employee(null, "CUS002", "林曉真");
            Set<Employee> set1 = new HashSet<>(Arrays.asList(emp1, emp2));
            Department dept1 = new Department(null, "CUS_A", "客戶服務部", set1);
            // -----
            Employee emp3 = new Employee(null, "ACC001", "劉芳");
            Employee emp4 = new Employee(null, "ACC002", "張君雅");
            Employee emp5 = new Employee(null, "ACC003", "陳淑芳");
            Set<Employee> set2 = new HashSet<>(Arrays.asList(emp3, emp4, emp5));
            Department dept2 = new Department(null, "ACC_A", "會計部", set2);
            // -----
        }
    }
}
```



# O2MMain01\_Uni\_Insert.java

..

```
Employee emp6 = new Employee(null, "ENG001", "莊明");
Set<Employee> set3 = new HashSet<>(Arrays.asList(emp6));
Department dept3 = new Department(null, "ENG_A", "工程部", set3);
SessionFactory sessionFactory = null;
Session session = null;
Transaction tx = null;
try{
// 建立SessionFactory物件
sessionFactory = HibernateUtils.getSessionFactory();
// 取出Session物件
session = sessionFactory.openSession();
System.out.println("得到Session物件");
//開啟交易
tx = session.beginTransaction();

// 因為Department類別有下列註釋，所以下面的for()迴圈可以省略
// @OneToMany(cascade=CascadeType.ALL)
//      不能使用(cascade=CascadeType.PERSIST)
//      for(Employee employee : set1){
//          session.save(employee);
//      }
//      for(Employee employee : set2){
//          session.save(employee);
//      }
```

# O2MMain01\_Uni\_Insert.java

...

```
//      for(Employee employee : set3){
//          session.save(employee);
//      }

//Save the Model objects
session.save(dept1);
session.save(dept2);
session.save(dept3);
//Commit transaction
System.out.println("=====");
tx.commit();
session.close();
}catch(Exception e){
    System.out.println("發生例外: "+e.getMessage());
    e.printStackTrace();
}finally{
    if(!sessionFactory.isClosed()){
        System.out.println("關閉SessionFactory");
        sessionFactory.close();
    }
}
}
```

# O2MMain01\_Uni\_Query.java .

---

```
// 摘要印出
SessionFactory sessionFactory = null;
Session session = null;
Transaction tx = null;
try {
    // 建立SessionFactory物件
    sessionFactory = HibernateUtils.getSessionFactory();
    // 取出Session物件
    session = sessionFactory.openSession();
    System.out.println("得到Session物件");
    // 開啟交易
    tx = session.beginTransaction();
    // 查詢特定的Department物件，在找出其內所有Employees
    System.out.println("查詢特定的Department物件，在找出其內所有Employees:");
    Department d1 = session.get(Department.class, 1);
    if (d1 != null) {
        for (Employee emp : d1.getEmployees()) {
            System.out.println("發現一個員工，id=" + emp.getId() + ", 姓名: " + emp.getName());
        }
    } else {
        System.out.println("查無資料");
    }
}
System.out.println("=====");
```

# O2MMain01\_Uni\_Query.java ..

---

```
// 查詢特定的Employee物件，由它找出對應的Department
System.out.println("無法由特定的Employee物件找出對應的Department，因為Employee物件內沒有"
    + "存放Department類別的物件");
tx.commit();
session.close();
} catch (Exception e) {
    System.out.println("發生例外：" + e.getMessage());
    e.printStackTrace();
} finally {
    if (!sessionFactory.isClosed()) {
        System.out.println("關閉SessionFactory");
        sessionFactory.close();
    }
}
}
```

# Department.java

```
package ch05.one2many._01.anno.model;
// 省略所有import敘述
@Entity
@Table(name = "ch05_om1_Department_UNI")
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "dept_id")
    private Integer id;
    private String deptCode;
    private String deptName;
    // 如果省略cascade={CascadeType.ALL}，Hibernate不會代為儲存多筆Employee物件，必須程式自行儲存
    // orphanRemoval只能用在@OneToMany與@OneToOne上
    @OneToMany(cascade = CascadeType.PERSIST, orphanRemoval = true)
    // @JoinColumn說明多方表格的fk_dept_id欄位為外鍵欄位(Employee類別對應的表格有fk_dept_id)，對照的主鍵
    // 為一方表格的dept_id欄
    // Employee(多方)類別中可以定義，也可以不定義對應外鍵欄位的屬性
    @JoinColumn(name = "fk_dept_id", referencedColumnName = "dept_id")
    private Set<Employee> employees = new LinkedHashSet<>();
    //
    public Department(Integer id, String deptCode, String deptName, Set<Employee> employees) {
        this.id = id;
        this.deptCode = deptCode;
        this.deptName = deptName;
        this.employees = employees;
    }
}
// 省略其餘敘述
```

# Employee.java

---

```
@Entity
@Table(name="ch05_om1_Employee_UNI")
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String employeeId;
    private String name;
    // @ManyToOne
    // private Department dept; // 由於缺少這個屬性，所以無法由Employee找到對應的Department

    public Employee() {
    }
    public Employee(Integer id, String employeeId, String name) {
        super();
        this.id = id;
        this.employeeId = employeeId;
        this.name = name;
    }

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    // 省略其餘敘述
}
```

# orphanRemoval = true

---

- 此屬性只能用於@OneToOne與@OneToMany
- 某個部門有很多位員工，若將其個員工由部門的 Set/List 物件中移出，Hibernate稱這樣的物件為孤兒(orphan)。  

```
Set<Employee> set = d1.getEmployees();  
set.remove(empRemove); // empRemove：某位員工
```
- 如果@OneToMany的orphanRemoval屬性設為 true，則此被移出員工對應的紀錄會於程式執行tx.commit()時，由Hibernate下達DELETE FROM的SQL命令刪除。

# 刪除orphan對應紀錄的SQL命令

---

Hibernate:

update

ch05\_om1\_Employee\_UNI

set

fk\_dept\_id=null

where

fk\_dept\_id=?

and id=?

Hibernate:

delete

from

ch05\_om1\_Employee\_UNI

where

id=?



# 程式片段

---

```
System.out.println("查詢特定的Department物件，在找出其內所有Employees:");
Employee empRemove = null;
Department d1 = session.get(Department.class, 1);
if (d1 != null) {
    for (Employee emp : d1.getEmployees()) {
        System.out.println("發現一個員工，id=" + emp.getId() + ", 姓名:" + emp.getName());
        if (emp.getId() == 2) {
            empRemove = emp;
            System.out.println("發現標的員工，id=" + empRemove.getId() + ", 姓名:" +
empRemove.getName());CascadeType.REMOVE
        }
    }
} else {
    System.out.println("查無資料");
}
Set<Employee> set = d1.getEmployees();
set.remove(empRemove);
System.out.println("=====");
tx.commit();
session.close();
```

# CascadeType.REMOVE

---

- 當執行`session.delete(dept1);` 準備將部門物件對應的紀錄由表格中刪除時，其內的所有員工也會從表格中被一併刪除。

//開啟交易

```
tx = session.beginTransaction();
```

```
Department dept1 = session.get(Department.class, 5);
```

```
System.out.println("1.-----");
```

```
session.delete(dept1);
```

```
System.out.println("2.-----");
```

//Commit transaction

```
tx.commit();
```

# 雙向一對多

- 雙向一對多:由部門(Department，一方)找出該部門內所有員工(Employee，多方)，也可以由員工找到所屬之部門。
  - 口訣：一對多，一方有個儲存多方物件參考的List/Set物件，簡稱『一方有個多』
- 只要部門(Department)內含有一個能儲存多方物件(員工類別)的List/Set物件，程式就可以由部門找到其內的所有員工。
- 以資料庫的觀點而言就是在Employee類別(多方)對應的表格內增加一個外鍵欄位，此外鍵欄位儲存Employee物件所屬之部門物件(一方)對應之紀錄的主鍵，Hibernate就可以由部門(Department)找出其內所有員工(Employee)。

# 雙向一對多的施行步驟

步驟 1	為永續類別加上應有的註釋(@Entity, @Table, @Id, ....)
步驟 2	由於程式的需求為『由部門(Department)找出其內的員工(Item)』，因此在Department類別內定義一個儲存Employee (多方)物件參考的List/Set型態的實例變數。 <code>private Set&lt;Employee&gt; employees = new LinkedHashSet&lt;&gt;();</code>
步驟 3	在此變數前加上 <code>@OneToMany(mappedBy = "dept")</code> mappedBy屬性表示本類別(Department)對應之表格並未含有可維護記錄關聯的外鍵，但在對照類別(Employee)內的dept屬性中有外鍵的相關資訊
步驟 4	在Employee類別內定義一個儲存Department (一方)物件參考的實例變數。
步驟 5	dept屬性前加上@ManyToOne，Many代表本類別(Employee)， "One"代表對照類別(Department)。
步驟 6	說明本類別(Employee)對應之表格內的外鍵名稱 <code>@JoinColumn(name="FK_dept_id")</code>

// 摘要印出

```
public class O2MMain04_Bi_Insert {

    public static void main(String[] args) {
        EmployeeBI emp1 = new EmployeeBI(null, "GAM001", "劉敏珍-BI");
        EmployeeBI emp2 = new EmployeeBI(null, "GAM002", "湯元泰-BI");
        Set<EmployeeBI> set1 = new HashSet<>(Arrays.asList(emp1, emp2));
        DepartmentBI dept1 = new DepartmentBI(null, "GAM_A", "遊戲部-BI", set1);
        emp1.setDept(dept1);
        emp2.setDept(dept1);
        // -----
        EmployeeBI emp3 = new EmployeeBI(null, "RES001", "林信民-BI");
        EmployeeBI emp4 = new EmployeeBI(null, "RES002", "吳雅芳-BI");
        EmployeeBI emp5 = new EmployeeBI(null, "RES003", "陳智勝-BI");
        Set<EmployeeBI> set2 = new HashSet<>(Arrays.asList(emp3, emp4, emp5));
        DepartmentBI dept2 = new DepartmentBI(null, "RES_A", "餐飲部-BI", set2);
        emp3.setDept(dept2);
        emp4.setDept(dept2);
        emp5.setDept(dept2);
        // -----
        EmployeeBI emp6 = new EmployeeBI(null, "TRA001", "莊淑芬-BI");
        Set<EmployeeBI> set3 = new HashSet<>(Arrays.asList(emp6));
        DepartmentBI dept3 = new DepartmentBI(null, "ENG_A", "旅遊部-BI", set3);
        emp6.setDept(dept3);
    }
}
```

```
SessionFactory sessionFactory = null;
Session session = null;
Transaction tx = null;
try {
    // 建立SessionFactory物件
    sessionFactory = HibernateUtils.getSessionFactory();
    // 取出Session物件
    session = sessionFactory.getCurrentSession();
    System.out.println("得到Session物件");
    // 開啟交易
    tx = session.beginTransaction();
    // Save the Model objects
    session.persist(dept1);
    session.persist(dept2);
    session.persist(dept3);
    // Commit transaction
    tx.commit();
    session.close();
    System.out.println("部門1 Id=" + dept1.getId());
    System.out.println("部門2 Id=" + dept2.getId());
    System.out.println("部門3 Id=" + dept3.getId());

} catch (Exception e) {
```

```
        System.out.println("發生例外: " + e.getMessage());
        e.printStackTrace();
    } finally {
        if (!sessionFactory.isClosed()) {
            System.out.println("關閉SessionFactory");
            sessionFactory.close();
        }
    }
}
```

# DepartmentBI.java

---

```
package ch05.one2many._04.anno.model;

// 省略import敘述

@Entity
@Table(name = "ch05_om4_Department_BI")
public class DepartmentBI {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "dept_id")
    private Integer id;
    private String deptCode;
    private String deptName;
    @OneToMany(mappedBy = "dept", fetch=FetchType.EAGER,
                cascade = { CascadeType.PERSIST }, orphanRemoval = false )
    private Set<Employee> employees = new LinkedHashSet<>();
    //
    public DepartmentBI(Integer id, String deptCode, String deptName, Set<Employee> employees) {
        this.id = id;
        this.deptCode = deptCode;
        this.deptName = deptName;
        this.employees = employees;
    }
    // 省略剩餘敘述
```



# EmployeeBI.java

■

```
package ch05.one2many._04.anno.model;
```

```
// 省略import敘述
```

```
@Entity
```

```
@Table(name="ch05_om4_Employee_BI")
```

```
public class EmployeeBI {
```

```
    @Id
```

```
    @GeneratedValue(strategy=GenerationType.IDENTITY)
```

```
    private Integer id;
```

```
    private String employeeId;
```

```
    private String name;
```

```
    @ManyToOne // 多對一，多方(Employee類別)內有個儲存一方(Department類別)物件參考的實例變數
```

```
    @JoinColumn(name="fk_dept_id", nullable=false)
```

```
    private Department dept; // 由於缺少這個屬性，所以無法由Employee找到對應的Department
```

```
    public EmployeeBI() {
```

```
    }
```

```
    public EmployeeBI(Integer id, String employeeId, String name) {
```

```
        super();
```

```
        this.id = id;
```

```
        this.employeeId = employeeId;
```

```
        this.name = name;
```

```
    }
```

# 單向多對一

- 單向多對一：由員工(Employee)找出他的雇主(Employer)，但無法由雇主找到員工。
  - － 口訣：多對一，多方有個儲存一方物件參考實例變數，簡稱『多方有個一』
  - － 只要員工類別有一個儲存雇主物件的實例變數，程式就可以由員工找到其雇主。但是由於是單向(Unidirectional)的多對一，所以一方不能含有多方的Set物件，否則就會形成雙向(Bidirectional)。
  - － 兩個表格的關聯為於多方(員工)類別中加入能儲存一方的物件參考。
- 以資料庫的觀點而言就是在員工(Employee)類別對應的表格內增加一個外鍵欄位，此欄位儲存該員工物件所屬之雇主的主鍵，Hibernate就可以由員工找出雇主。

# 單向多對一的施行步驟

步驟 1	為永續類別加上應有的註釋(@Entity, @Table, @Id, ....)
步驟 2	由於程式的需求為「由員工找到其雇主」，因此在Employee類別內定義一個儲存Employer物件參考的實例變數，即 <pre>private Employer employer;</pre>
步驟 3	在此實例變數前加上 <pre>@ManyToOne(cascade=CascadeType.ALL) public Employer getEmployer() {     return employer; }</pre> "Many"表示本類別(Employee)，"One"表示對照類別(Employer)。
步驟 4	Employer類別無需特別的註釋

# 單向多對一 M2O\_UNI\_Insert\_Main01.java

---

```
// 摘要印出
EmployeeDAO dao = new EmployeeDAO();
EmployerUNI emperA = new EmployerUNI("劉瑪莉老闆"); // 雇主類別
EmployeeUNI empeeA1 = new EmployeeUNI("劉小明"); // 員工類別
emperA1.setEmployer(emperA);
dao.save(empeeA1);
EmployeeUNI empeeA2 = new EmployeeUNI("劉美華"); // 員工類別
emperA2.setEmployer(emperA);
dao.save(empeeA2);

EmployerUNI emperB = new EmployerUNI("林芳華老闆"); // 雇主類別
EmployeeUNI empeeB1 = new EmployeeUNI("林偉明"); // 員工類別
emperB1.setEmployer(emperB);
dao.save(empeeB1);
EmployeeUNI empeeB2 = new EmployeeUNI("林世光"); // 員工類別
emperB2.setEmployer(emperB);
dao.save(empeeB2);

EmployerUNI emperC = new EmployerUNI("黃河新老闆"); // 雇主類別
EmployeeUNI empeeC1 = new EmployeeUNI("黃天南"); // 員工類別
emperC1.setEmployer(emperC);
dao.save(empeeC1);

HibernateUtils.close();
```

# EmployeeUNI.java

---

```
package ch05.many2one._01.anno.model;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.ForeignKey;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "ch05_mo1_Employee_UNI")
public class EmployeeUNI {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String employeeName;
    // 只有@OneToOne, @OneToMany 可以使用 orphanRemoval = true
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "employer_id" , foreignKey=@ForeignKey(name = "fk_eee_eer"))
    private EmployerUNI employer;
```

# EmployerUNI.java

---

```
package ch05.many2one._01.anno.model;
import javax.persistence.*;
//一個Employer(雇主)對應多個Employee(員工)
@Entity
@Table(name="ch05_mo1_Employer_UNI")
public class EmployerUNI {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String employerName;

    public EmployerUNI() { }

    public EmployerUNI(String employerName) {
        this.employerName = employerName;
    }
    // 省略Getter/Setter
}
```

# 雙向多對一

- 雙向多對一與雙向一對多的做法完全相同
- 雙向多對一：一個戶籍地址(AddressBi)可以包含許多人(PersonBi)，而每個人只有一個戶籍地址。
- 可由PersonBi(多方)找到AddressBi(一方)，因此PersonBi必須定義一個能儲存AddressBi之物件參考的實例變數。

```
private AddressBi addressBi
```

- 由於是雙向，因此也可由AddressBi(一方)找到PersonBi(多方)，因此AddressBi必須定義一個能儲存多個PersonBi物件的Set<PersonBi>型別的實例變數。

```
Set<PersonBi> set = new HashSet<>();
```

# 雙向多對一

■ ■

- 於PersonBi類別中，private AddressBi addressBi; 實例變數前加上

```
@ManyToOne(cascade=CascadeType.ALL)
@JoinColumn(name="fk_address_id")
public AddressBi getAddressBi() {
    return addressBi;
}
```

加上@ManyToOne註釋的目的：

1. 讓Hibernate知道address是表示類別之間的Association而非一個欄位
2. 讓Hibernate在Person\_Table中建立一個做為外鍵的欄位，欄位名稱可以是自定(透過@JoinColumn)

或由Hibernate依照預設規則決定欄位名稱(外鍵之來源表格名稱 + "\_" + 來源表格的主鍵名稱)

- 於AddressBi類別中，Set<PersonBi> set = new HashSet<>(); 實例變數前加上

```
@OneToMany(mappedBy = "address", cascade = CascadeType.ALL)
public Set<PersonBi> getSet() {
    return set;
}
```



# 雙向多對一-M2OMain04\_Bi\_anno\_Insert.java -1

---

```
// 摘要印出
AddressBi ad1 = new AddressBi("台北市松山區松山路101號");
AddressBi ad2 = new AddressBi("台北市士林區中山北路七段135號");
PersonBi p1 = new PersonBi("張君雅(松山區)");
p1.setAddressBi(ad1);
PersonBi p2 = new PersonBi("劉麗芳(松山區)");
p2.setAddressBi(ad1);
PersonBi p3 = new PersonBi("徐衛國(士林區)");
p3.setAddressBi(ad2);
PersonBi p4 = new PersonBi("林曉芳(士林區)");
p4.setAddressBi(ad2);
SessionFactory factory = HibernateUtils.getSessionFactory();
Session session = factory.openSession();
Transaction tx = session.beginTransaction();
try{
    //    session.save(ad1); // 如果有加Cascade時可省略
    session.save(p1);
    session.save(p2);
    System.out.println("-----");
    session.save(p3);
    session.save(p4);
    tx.commit();
} catch(Exception e){
    // 以下省略
}
```

# AddressBi.java

---

```
// 摘要印出
@Entity
@Table(name = "ch05_mo2_AddressBi_Table")
public class AddressBi {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer addressId;
    private String name;

    //mappedBy: 說明本類別並未含有能夠表示關聯的資訊，此項資訊位於PersonBi類別的
    //address性質中
    @OneToMany(mappedBy = "addressBi", cascade = CascadeType.ALL)
    Set<PersonBi> set = new HashSet<>();

    public AddressBi() {
    }

}
```

# PersonBi.java

---

```
// 摘要印出
@Entity
@Table(name = "ch05_mo2_PersonBi_Table")
public class PersonBi {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer personId;
    private String name;
    // 加上@ManyToOne註釋的目的：
    // 1. 讓Hibernate知道address是表示類別之間的Association而非一個欄位
    // 2. 讓Hibernate在Person_Table中建立一個做為外鍵的欄位，欄位名稱可以是自定(透過@JoinColumn)
    // 或由Hibernate依照預設規則決定欄位名稱(外鍵之來源表格名稱 + "_" + 來源表格的主鍵名稱)
    @ManyToOne(cascade = CascadeType.ALL)
    // 加上@JoinColumn註釋的目的：指定Person_Table中，外鍵的欄位名稱
    @JoinColumn(name = "fk_address_id")
    private AddressBi addressBi;

    public PersonBi() {
    }
}
```

# 雙向多對多

---

- 雙向多對多：由作者(Author)找出其撰寫的書籍(Book)，也可以由書籍找到作者。
  - － 口訣：多對多，雙方都有個可以儲存對方物件參考的List/Set型態的變數，簡稱『多方有個多』

# 雙向多對多的施行步驟

步驟 1	為永續類別加上應有的註釋(@Entity, @Table, @Id, ....)
步驟 2	Book類別: 定義一個儲存Author物件參考的Set型態的變數, 即 <code>private Set&lt;Author&gt; authors = new HashSet&lt;Author&gt;(0);</code>
步驟 3	在authors變數前加入下列註釋 <code>@ManyToMany(cascade = CascadeType.ALL)</code> <code>@JoinTable(name = "author_book",</code> <code>joinColumns = {</code> <code>@JoinColumn(name = "BOOK_ID", nullable = false, updatable = false)</code> <code>},</code> <code>inverseJoinColumns = {</code> <code>@JoinColumn(name = "AUTHOR_ID", nullable = false, updatable = false)</code> <code>}</code> <code>)</code> <code>private Set&lt;Author&gt; authors = new HashSet&lt;Author&gt;(0);</code>
步驟 4	Author類別 定義一個儲存Book物件參考的Set型態的變數, 即 <code>private Set&lt;Book&gt; books = new HashSet&lt;Book&gt;(0);</code> 並在此變數前加上下列註釋 <code>@ManyToMany(fetch = FetchType.LAZY, mappedBy = "authors")</code> <code>private Set&lt;Book&gt; books = new HashSet&lt;Book&gt;(0);</code>

```
SessionFactory sessionFactory = null;
Session session = null;
Transaction tx = null;
try {
    // 建立SessionFactory物件
    sessionFactory = HibernateUtils.getSessionFactory();
    // 取出Session物件
    session = sessionFactory.openSession();
    System.out.println("得到Session物件");
    // 開啟交易
    tx = session.beginTransaction();
    Book book1 = new Book("快樂學JSP");
    Book book2 = new Book("Hibernate企業實戰");
    Book book3 = new Book("Spring精典應用");
    Author a1 = new Author("張君雅(J,S)");
    Author a2 = new Author("劉翰林(J,S)");
    Author a3 = new Author("黃美智(H,S)");

    Set<Author> set_s = new HashSet<Author>();
    set_s.add(a1);
    set_s.add(a2);
    set_s.add(a3);
```

# 雙向多對多 M2MMain01\_Anno\_Insert.java -2

---

```
Set<Author> set_j = new HashSet<Author>();
    set_j.add(a1);
    set_j.add(a2);

    Set<Author> set_h = new HashSet<Author>();
    set_h.add(a3);

    book1.setAuthors(set_j);
    book2.setAuthors(set_h);
    book3.setAuthors(set_s);

    session.save(book1);
    session.save(book2);
    session.save(book3);
    tx.commit();
    System.out.println("程式執行完畢");
} catch (Exception e) {
    System.out.println("Exception occurred. " + e.getMessage());
    e.printStackTrace();
} finally {
    if (!sessionFactory.isClosed()) {
        System.out.println("Closing SessionFactory");
        sessionFactory.close();
    }
}
```

# Author.java

---

```
package ch05.many2many._01.anno.model;

import java.util.*;
import javax.persistence.*;
@Entity(name = "ch05_mm1_Author")
@Table(name = "ch05_mm1_Author")
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "AUTHOR_ID", unique = true, nullable = false)
    private Integer authorId;

    @Column(name = "AUTHOR_Name")
    private String authorName;

    @ManyToMany(mappedBy = "authors")
    private Set<Book> books = new HashSet<Book>(0);

    public Author() { }

    public Author(String authorName) {
        this.authorName = authorName;
    }
}
```



# Book.java

---

```
package ch05.many2many._01.anno.model;
import static javax.persistence.GenerationType.IDENTITY;
// 省略import敘述
@Entity(name = "ch05_mm2_Book")
@Table(name = "ch05_mm2_Book")
public class Book {
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "BOOK_ID")
    private Integer bookId;
    @Column(name = "book_name")
    private String bookName;
    @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
    @JoinTable(name = "ch05_mm1_author_book",
        joinColumns = { // 在Join Table中，儲存本類別之主鍵值的外鍵欄位名稱
            @JoinColumn(name = "FK_BOOK_ID", referencedColumnName = "BOOK_ID",
                foreignKey=@ForeignKey(name = "fkc_ba_boo"))
        },
        inverseJoinColumns = { // 在Join Table中，儲存對應對照類別之主鍵值的外鍵欄位名稱
            @JoinColumn(name = "FK_AUTHOR_ID", referencedColumnName = "AUTHOR_ID",
                foreignKey=@ForeignKey(name = "fkc_ba_aut"))
        }
    )
    private Set<Author> authors = new HashSet<Author>(0);
    // 省略剩餘敘述
```



---

## 六、Hibernate Query Language(HQL)

# Hibernate Query Language

---

- 是一種功能強大，語法類似SQL的查詢語言
- 查詢語言操作的對象是類別與類別的性質而非表格與表格的欄位，而Hibernate會將HQL轉為對表格與欄位的操作，可以進行表格資料的增刪改查。
- HQL語言不分大小寫，但是Java類別名稱與欄位名稱有區分大小寫，而其他關鍵字不區分大小寫
- 是Hibernate官方推薦的查詢語言

# Query 介面

---

- Hibernate的HQL都必須經由Query介面提供的方法執行。
- Session介面的createQuery(String HQL)可產生Query物件。
- Query介面於Hibernate 5.1版前位於org.hibernate套件下，即org.hibernate.Query。Hibernate 5.2版起推出位於org.hibernate.query套件下新版的Query介面，即org.hibernate.query.Query。
- 舊的Query介面於5.2版起作廢，計畫於Hibernate 6.0移除舊版的Query介面。

# 執行HQL的步驟

---

- 產生Query物件：  
Query query = session.createQuery(hql);
  - 由Session物件的createQuery(hql)產生Query物件。
- 進行前置作業：替HQL參數付值、設定讀取物件的範圍  
query.setParameter("參數名稱1",參數值1)  
    .setParameter("參數名稱2",參數值2)  
    ...  
    .setFirstResult(5)     // 由第六筆開始讀  
    .setMaxResults(3)     // 最多讀三筆
- 執行HQL  
    query.getResultList() 或  
    query.getSingleResult() 或  
    query.executeUpdate()

# Query介面常用的方法

---

- `getResultList()`：傳回`java.util.List<R>`的物件
  - 用以查詢0或多筆物件
- `getSingleResult()`：傳回單一物件
  - 查詢正好為1筆的物件，傳回0筆或多筆都會丟出例外
    - 傳回0筆物件會丟出 `javax.persistence.NoResultException`
    - 傳回多筆物件會丟出 `org.hibernate.NonUniqueResultException`
- `setParameter("參數的名稱", 參數的值)`：設定HQL內的參數
  - 在HQL內可定義多個以冒號(:)開頭的變數，這些變數都要用不同的`setParameter("參數的名稱", 參數的值)`設值。
- `executeUpdate()`：執行UPDATE/DELETE/INSERT HQL敘述
- `setFirstResult(s)`：說明要由第幾筆物件開始讀取(0開頭)
- `setMaxResults(len)`：說明要讀多少筆物件

# 樣本資料

---

- Employee表格

張君雅|27400|1978-07-18|1  
黃明和|35000|1969-08-18|2  
劉麗芳|49000|1985-12-25|2  
朱定華|35000|1984-05-03|1  
李明珊|28000|1981-07-16|2  
譚自強|39000|1989-09-30|1  
林惠明|35000|1984-02-24|1  
侯曉君|47500|1985-10-15|2  
何曉君|47400|1980-11-25|3  
劉芬芬|29850|1975-12-14|3  
楊光華|40200|1973-03-23|3  
王寶生|41600|1986-04-29|3  
胡曉明|36700|1989-08-31|3  
江雪琳|47000|1981-01-11|2

- EmployeeA表格

丁明|27400|1970-12-25|1  
王芳|35000|1964-03-18|2  
刁華|49000|1975-02-17|2  
卜珍|29000|1975-12-25|2

- 先執行**ch06.\_init.EmployeeInitialization.java**匯入樣本資料



# HQL查詢傳回值的型態

---

- 查詢標的為類別，FROM之後必須提供 **Entity Name**
  - "FROM **Employee** e"
    - 傳回值的型態為List<Employee>
- 查詢標的為單一性質
  - "SELECT e.name FROM **Employee** e"
    - 傳回值的型態為List<String>
  - "SELECT e.salary FROM **Employee** e"
    - 傳回值的型態為List<Integer>
- 查詢標的為多個性質
  - "SELECT e.name, e.salary, e.birthday FROM **Employee** e"
    - 傳回值的型態為List<Object[]>

# HQL的FROM子句

- 透過Query物件進行查詢，程式可先設定查詢參數，透過相關的方法將指定的參數值填入：

**HQL: "FROM Employee"; "FROM Employee AS e"; "FROM Employee e";**

位於 [ch06.HibernateQueryExercise01.java](#)

```
public static void main(String[] args) {  
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();  
    String hql = "FROM Employee e";  
    Transaction tx = null;  
    try {  
        tx = session.beginTransaction();  
        List<Employee> emps = session.createQuery(hql).getResultList();  
        for (Employee e : emps) {  
            System.out.println(e);  
        }  
        tx.commit();  
    } catch (Exception e) {  
        if (tx != null) {  
            tx.rollback();  
        }  
    }  
}
```

# HQL的SELECT子句-查詢單一性質

---

- 查詢單一性質，傳回值為List<?>

**HQL: "SELECT e.name FROM Employee e"**

[位於 ch06.HibernateQueryExercise02.java](#)

```
public static void main(String[] args) {  
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();  
    String hql = "SELECT e.name FROM Employee e";  
    Transaction tx = null;  
    try {  
        tx = session.beginTransaction();  
        List<String> names = session.createQuery(hql).getResultList();  
        for(String name : names) {  
            System.out.println(name);  
        }  
        tx.commit();  
    } catch(Exception e) {  
        if (tx != null) {  
            tx.rollback();  
        }  
    }  
}
```

# HQL的SELECT子句-查詢單一性質

---

- 查詢單一性質，傳回值為List<?>

**HQL: "SELECT e.salary FROM Employee e"**

[位於 ch06.HibernateQueryExercise03.java](#)

```
public static void main(String[] args) {  
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();  
    String hql = "SELECT e.salary FROM Employee e";  
    Transaction tx = null;  
    try {  
        tx = session.beginTransaction();  
        List<Integer> salaries = session.createQuery(hql).getResultList();  
        for(Integer salary : salaries) {  
            System.out.println(salary);  
        }  
        tx.commit();  
    } catch(Exception e) {  
        if (tx != null) {  
            tx.rollback();  
        }  
    }  
}
```

# HQL的SELECT子句-查詢多個性質

- 查詢多個性質，傳回值為List<Object[]>

**HQL = "select e.name, e.salary, e.birthday FROM Employee e";**

位於 [ch06.HibernateQueryExercise04.java](#)

```
public static void main(String[] args) {  
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();  
    String hql = "SELECT e.salary, e.name, e.birthday FROM Employee e";  
    Transaction tx = null;  
    try {  
        tx = session.beginTransaction();  
        List<Object[]> list = session.createQuery(hql, Object[].class).getResultList();  
        for(Object[] oa : list) {  
            System.out.println(oa[0] + ", " + oa[1] + ", " + oa[2]);  
        }  
        tx.commit();  
    } catch(Exception e) {  
        if (tx != null) {  
            tx.rollback();  
        }  
    }  
}
```

# HQL的WHERE子句

---

**hq.query5("FROM Employee e WHERE e.name = '張君雅' and e.depId = 1");**

1 張君雅 27400 1978-12-25 00:00:00.0 1

**hq.query5("FROM Employee e WHERE e.salary >= 30000");**

2 黃明和 35000 1969-03-18 00:00:00.0 2

3 劉麗芳 49000 1985-12-25 00:00:00.0 2

4 朱定華 35000 1974-05-03 00:00:00.0 1

**hq.query5("FROM Employee e WHERE e.salary >= 30000 and e.name like '黃%');**

2 黃明和 35000 1969-03-18 00:00:00.0 2

# HQL的ORDER BY子句

---

[位於 ch06.HibernateQueryExercise05.java](#)

```
public static void main(String[] args) {  
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();  
    String hql = "FROM Employee e WHERE e.salary >= :sal ORDER BY e.salary ASC, e.id ASC";  
    Transaction tx = null;  
    try {  
        tx = session.beginTransaction();  
        List<Employee> emps = session.createQuery(hql, Employee.class)  
            .setParameter("sal", 29000).getResultList();  
        for(Employee e : emps) {  
            System.out.println(e);  
        }  
        tx.commit();  
    } catch(Exception e) {  
        if (tx != null) {  
            tx.rollback();  
        }  
    }  
}
```

# HQL的聚合函數

---

[位於 ch06.HibernateQueryExercise06.java](#)

```
public static void main(String[] args) {
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();
    String hql = "SELECT SUM(e.salary), MAX(e.salary) FROM Employee e ";
    Transaction tx = null;
    try {
        tx = session.beginTransaction();

        List<Object[]> emps = session.createQuery(hql, Object[].class)
            .getResultList();
        for(Object[] oa : emps) {
            System.out.println(oa[0] + ", " + oa[1]);
        }
        tx.commit();
    } catch(Exception e) {
        if (tx != null) {
            tx.rollback();
        }
    }
}
```



# HQL的GROUP BY子句

---

[位於 ch06.HibernateQueryExercise07.java](#)

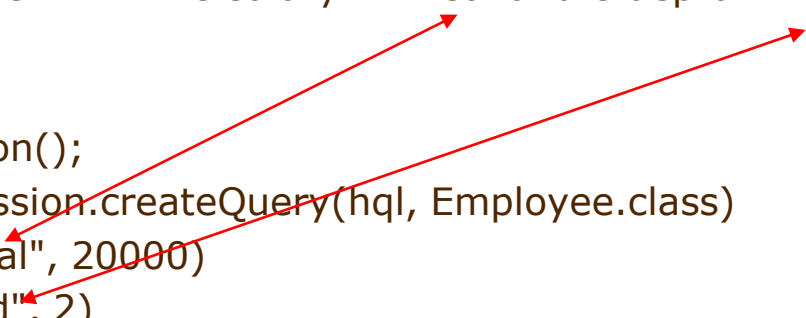
```
public static void main(String[] args) {
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();
    String hql = "SELECT e.depId, SUM(e.salary), MAX(e.salary) " +
        " FROM Employee e GROUP BY e.depId";
    Transaction tx = null;
    try {
        tx = session.beginTransaction();

        List<Object[]> emps = session.createQuery(hql)
            .getResultList();
        for(Object[] oa : emps) {
            System.out.println(oa[0] + ", " + oa[1]+ ", " + oa[2]);
        }
        tx.commit();
    } catch(Exception e) {
        if (tx != null) {
            tx.rollback();
        }
    }
}
```

# Named Parameters Query

位於 [ch06.HibernateQueryExercise08.java](#)

```
public static void main(String[] args) {
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();
    String hql = "FROM Employee e WHERE e.salary >= :sal and e.depId = :id";
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        List<Employee> emps = session.createQuery(hql, Employee.class)
            .setParameter("sal", 20000)
            .setParameter("id", 2)
            .getResultList();
        for(Employee e : emps) {
            System.out.println(e);
        }
        tx.commit();
    } catch (Exception e) {
        if (tx != null) {
            tx.rollback();
        }
    }
}
```



The diagram illustrates the mapping of named parameters in the HQL query to their values in the Java code. Red arrows point from the parameter names in the HQL string to the values provided in the `setParameter` calls:

- Arrow from `:sal` in the HQL to `20000` in `setParameter("sal", 20000)`.
- Arrow from `:id` in the HQL to `2` in `setParameter("id", 2)`.

# Positional Paramters Query

Deprecated

//Using Positional Paramters

```
public List<Employee> query9(){
    Session session = HibernateUtils.getSessionFactory().openSession();
    String hql = "FROM Employee e WHERE e.salary > ? and "
        + "birthday > ? order by salary desc";
    TypedQuery<Employee> query = session.createQuery(hql);
    query.setParameter(0, 25000);
    Date d = Date.valueOf("1979-12-01");
    query.setParameter(1, d);
    List<Employee> list = query.getResultList();
    session.close();
    for(Employee e : list){
        System.out.printf("%2d %6s %6d %24s %2d\n", e.getId(), e.getName(),
            e.getSalary(), e.getBirthday(), e.getDepId());
    }
    return list;
}
```

HQL不支援用問號當作位置參數

# UPDATE子句

---

位於 [ch06.HibernateQueryExercise09.java](#)

```
public static void main(String[] args) {
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();
    String hql = "UPDATE Employee e SET e.salary = e.salary + :sal, e.birthday = :birth"
        + " WHERE e.id = :empid";
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.createQuery(hql)
            .setParameter("sal", 3000)
            .setParameter("birth", java.sql.Date.valueOf("1982-1-25"))
            .setParameter("empid", 5)
            .executeUpdate();

        tx.commit();
    } catch (Exception e) {
        if (tx != null) {
            tx.rollback();
        }
    }
}
```

# DELETE子句

---

[位於 ch06.HibernateQueryExercise10.java](#)

```
public static void main(String[] args) {
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();
    String hql = "DELETE FROM Employee e WHERE e.id = :empid";
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.createQuery(hql)
            .setParameter("empid", 3)
            .executeUpdate();

        tx.commit();
    } catch (Exception e) {
        if (tx != null) {
            tx.rollback();
        }
    }
}
```

# 分頁

---

位於 [ch06.HibernateQueryExercise11.java](#)

```
public static void main(String[] args) {
    Session session = HibernateUtils.getSessionFactory().getCurrentSession();
    String hql = "FROM Employee e ORDER By e.id";
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        List<Employee> emps = session.createQuery(hql, Employee.class)
            .setFirstResult(4)
            .setMaxResults(2)
            .getResultList();
        for (Employee e : emps) {
            System.out.println(e);
        }
        tx.commit();
    } catch (Exception e) {
        if (tx != null) {
            tx.rollback();
        }
    }
}
```

# INSERT INTO子句

[位於 ch06.HibernateQueryExercise12.java](#)

```
public static void main(String[] args) {  
    Session session = HibernateUtils.getSessionFactory().openSession();  
    // 下列都是類別與性質名稱，注意大小寫  
    String hql = "INSERT INTO Employee(birthday, depId, name, salary)"  
        + "SELECT birthday, depId, name, salary FROM EmployeeA";  
    Transaction tx = null;  
    Query<?> query = null;  
    try {  
        tx = session.beginTransaction();  
        query = session.createQuery(hql);  
        query.executeUpdate();  
        tx.commit();  
    } catch (Exception e) {  
        if (tx != null)  
            tx.rollback();  
        e.printStackTrace();  
    }  
}
```

HQL的INSERT INTO不支援  
VALUES子句

# Hibernate Native Query

---

[位於 ch06.HibernateQueryExercise12.java](#)

```
public static void main(String[] args) {
    String sql = "SELECT * FROM ch06_EMPLOYEE";
    Session session = HibernateUtils.getSessionFactory().openSession();
    List<Object[]> list = null;
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        list = session.createNativeQuery(sql).list();
        tx.commit();
    } catch (Exception e) {
        if (tx != null)
            tx.rollback();
        throw new RuntimeException(e);
    }
    for(Object[] oa :list) {
        for(Object obj : oa) {
            System.out.print(obj+ " ");
        }
        System.out.println();
    }
}
```



---

## 七、附錄

# 建立SessionFactory物件 (限Hibernate 3.x)

---

```
private static SessionFactory buildSessionFactory() {  
    try {  
        // Hibernate 3 的寫法  
        return new Configuration().configure().buildSessionFactory();  
    } catch (Throwable ex) {  
        System.err.println("新建SessionFactory失敗, " + ex);  
        throw new ExceptionInInitializerError(ex);  
    }  
}
```

# 建立SessionFactory物件 (限Hibernate 4.0/4.1/4.2)

---

```
private static SessionFactory sessionFactory;  
private static ServiceRegistry serviceRegistry;  
  
private static SessionFactory buildSessionFactory() {  
    try {  
        // Hibernate 4.0, 4.1, 4.2 的寫法  
        Configuration configuration = new Configuration();  
        configuration.configure();  
        serviceRegistry = new ServiceRegistryBuilder().applySettings(  
            configuration.getProperties()).buildServiceRegistry();  
        sessionFactory = configuration.buildSessionFactory(serviceRegistry);  
        return sessionFactory;  
    } catch (Throwable ex) {  
        System.err.println("新建SessionFactory失敗, " + ex);  
        throw new ExceptionInInitializerError(ex);  
    }  
}
```

- 4.x 以後新增一個ServiceRegistry介面，Hibernate的任何配置與服務都要在ServiceRegistry內註冊後才會生效。

# 建立SessionFactory物件 (限Hibernate 4.3)

---

```
private static SessionFactory buildSessionFactory() {  
    try {  
        // Hibernate 4.3 的寫法  
        Configuration configuration = new Configuration();  
        configuration.configure("/config/hibernate.cfg.xml");  
        serviceRegistry = new StandardServiceRegistryBuilder()  
            .applySettings(configuration.getProperties()).build();  
        sessionFactory = configuration.buildSessionFactory(serviceRegistry);  
        return sessionFactory;  
    } catch (Throwable ex) {  
        System.err.println("新建SessionFactory失敗, " + ex);  
        throw new ExceptionInInitializerError(ex);  
    }  
}
```

# Mapping檔: member.hbm.xml(舊式)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <!-- 本檔案描述的類別(ch04.ex01.model.Member)與表格名稱(MemberExample) -->
  <!-- 所有欄位都需要在Member.java內準備Getter/Setter，缺一不可 -->
  <class name="ch04.ex01.model.Member" table="MemberExample">
    <!-- name: 類別內的性質(property)名稱，column: 表格內的欄位名稱 -->
    <!-- 表示當永續本類別的物件時，性質id的值會儲存到表格的PK欄位內 -->
    <id name="id" column="PK" type="java.lang.Integer">
      <generator class="native" />
    </id>
    <!-- 省略length屬性時，如果欄位型態為varchar，則預設值為255，如果欄位型態為int，則預設值為11 -->
    <!-- 省略type屬性時，Hibernate會用Reflection推算每個欄位的型態，如果欄位型態為varchar，則預設值為
    255，如果欄位型態為int，則預設值為11。 -->
    <!-- 表示當永續本類別的物件時，性質userId的值會儲存到表格的 account欄位內 -->
    <property name="userId" column="account" length="32" />
    <!-- 省略column屬性時，它的內含值與name屬性相同 -->
    <property name="password" length="32" />
    <property name="name" length="32" />
    <property name="email" length="64" />
    <property name="tel" length="64" />
    <property name="experience" />
  </class>
</hibernate-mapping>
```

# Mapping檔

- 描述永續類別與表格的映射資訊

- 類別名稱與表格名稱

- `<class name="ch04.ex01.model.Member" table="MemberExample">`

- 類別內必須要有一個唯一(不重覆)的性質(Property)，此性質可以對應到表格的主要鍵(Primary Key)。主要鍵與對應的性質要定義在<id>標籤內，非主要鍵欄位與對應的性質則定義在<property>標籤內。

- `<id name="id" column="PK" type="java.lang.Integer">`

- `<generator class="native" />`

- `</id>`

- **name:** 類別內的性質名稱，**column:** 表格內的欄位名稱，表示當永續本類別的物件時，性質id的值會儲存到表格的PK欄位內
    - **type:** 指定本欄位的型態，它不是Java型態，也不是SQL Type而是Hibernate定義的型態，詳見org.hibernate.Hibernate類別的定義。
    - `<generator class="native" />` 說明主鍵的產生方式，設定為"native"表示依使用之資料庫的主鍵生成方式產生主鍵值，例如使用MySQL資料庫，表格之主鍵會多一個屬性auto\_increment，每加入一筆紀錄後會自動遞增主鍵值。

# Mapping檔

..

```
<property name="userId" column="account" length="32" />
```

- 非主鍵欄位用<property>標籤定義。
- name: 類別內的性質名稱，column: 表格內的欄位名稱，表示當永續本類別的物件時，性質userId的值會儲存到表格的account欄位內。
- length="32": 欄位長度為32。省略length屬性時，如果欄位型態為varchar，則預設值為255。如果欄位型態為int，則預設值為11。
- 省略type屬性時，Hibernate會用Reflection推斷每個欄位的Hibernate型態。

---



---

# Hibernate 作業

# 練習說明

---

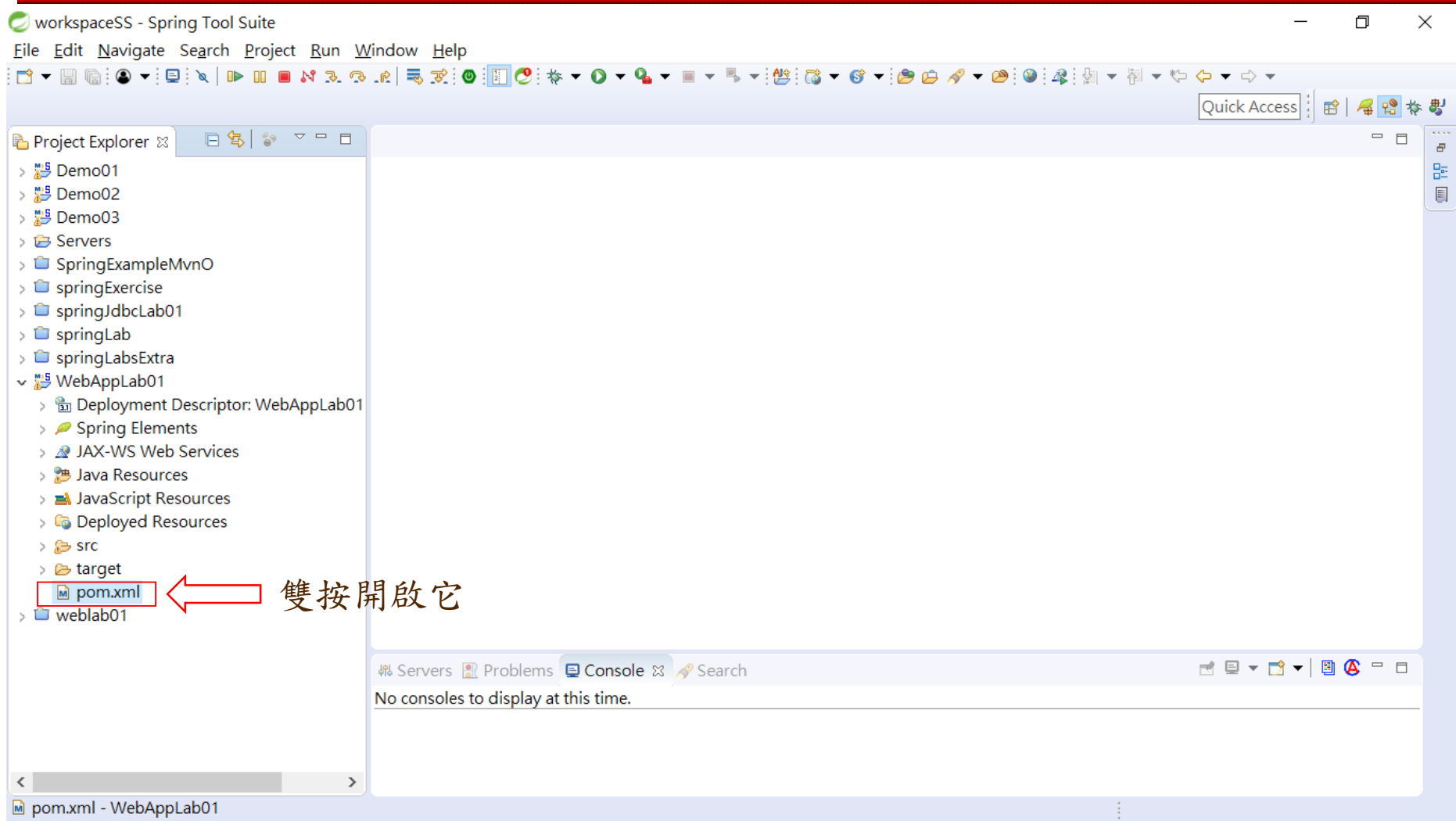
- WebAppLab01 專案內含一個可以執行的網路應用系統，它可對Member表格內的紀錄進行增、刪、改、查等基本操作。
- 此專案為MVC架構，以Servlet作為Controller，用JSP扮演View，以MemberService/MemberDAO 兩個JavaBean作為Model。它的MemberDAO是採用JDBC的技術存取資料庫。
- 請修改本專案完成下列工作：  
以Hibernate技術取代該專案原用之Jdbc技術存取資料庫內的資料。

# 編寫步驟

---

- 匯入專案
  - WebAppLab01\_題目\_MySQL.zip (MySQL)
  - WebAppLab01\_題目\_MsSQL.zip (SQL Server)
- 建立表格與初始資料
  - My SQL Ceate Table and Insert Into .sql (MySQL)
  - SQLServer Ceate Table and Insert Into .sql (SQL Server)
- 替pom.xml增加必要的<dependency>
- 創建hibernate.cfg.xml
- 修改DAO類別，用Hibernate框架提供的類別對表格內的紀錄進行增、刪、改、查等基本操作。
  - 編寫Hibernate組態檔
  - 修改相關的POJO類別，以便提供O/R Mapping的依據
  - 新建使用Hibernate技術，實作MemberDao介面的DAO類別。

# 替 pom.xml 加入必要的 <dependency>



# Hibernate-core

Maven Repository: org.hibernate

+

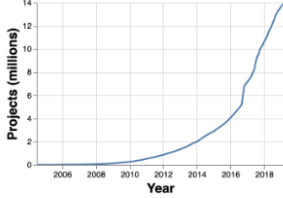
mvnrepository.com/artifact/org.hibernate/hibernate-core

Search for groups, artifacts, categories

Search

Categories | Popular | Contact Us

Indexed Artifacts (28.1M)



Popular Categories

Aspect Oriented

Actor Frameworks

Application Metrics

Build Tools

Bytecode Libraries

Command Line Parsers


Cache Implementations

Cloud Computing

Code Analyzers

Collections

Home » org.hibernate » hibernate-core



Hibernate Core Relocation

Hibernate's core ORM functionality

License	LGPL 2.1
Categories	Object/Relational Mapping
Tags	hibernate persistence relational mapping
Used By	3,885 artifacts

Note: This artifact was moved to:

org.hibernate.orm » hibernate-core » 6.0.1.Final

Central (279)

Atlassian (1)

Atlassian 3rdParty (4)

Atlassian 3rd-P Old (24)

Spring Lib Release (1)

Spring Plugins (4)

Redhat GA (61)

Redhat EA (20)

ImageJ Public (2)

Grails Core (5)

JCenter (23)

Geomajas (1)

TU-Darmstadt (1)

ICM (4)

Ads by Google

Stop seeing this ad

Why this ad?

在 這裡輸入文字來搜尋



下午 02:27  
2022/5/23

# 編寫Hibernate組態檔

---

- Hibernate組態檔(hibernate.cfg.xml)之位置乃相對於類別路徑的根目錄。
- Hibernate組態檔定義三類資訊：
  - － 連接資料庫所需的連線資訊，如帳號、密碼、連線字串等資訊。
  - － 說明類別與表格、屬性與欄位等OR Mapping相關資訊。
  - － 其他雜項資訊。
- 本練習的Hibernate組態檔已經寫好，請匯入。

# 在類別中加上OR Mapping的註釋

workspaceA - WebAppLab01/src/main/java/lab01/model/MemberBean.java - Spring Tool Suite 4

File Edit Source Refactor Navigate Search Project Run Window Help

Project Explorer

- ajax22mm
- HibernateExamplesMM22
- HibernateExamplesMM23
- HibernateExamplesMsSQL2019
- Servers
- SpringMVC\_CRUD22
- SpringRest22mm
- WebAppLab01
  - JAX-WS Web Services
  - src/main/java
    - lab01
      - controller
      - dao
      - model
        - MemberBean.java**
        - service
        - SystemUtils.java
  - src/main/resources
  - src/test/java
  - src/test/resources
  - Maven Dependencies
  - Apache Tomcat v9.0 [Apache Tomcat]
  - JRE System Library [JavaSE-11]

MemberBean.java

```
1 package lab01.model;
2
3 import java.io.Serializable;
4
5
6
7
8
9 public class MemberBean implements Serializable {
10     private static final long serialVersionUID = 1L;
11     private Integer id; // ObjectID
12     private String memberId; // 帳號
13     private String password; // 密碼
14     private String name; // 姓名
15     private String phone; // 電話
16     private java.util.Date birthday; // 生日
17     private java.sql.Timestamp registerDate; // 會員登錄日期
18     private double weight; // 體重
19
20     public MemberBean() {
21     }
```

MemberBean表達類別與表格的對應關係。

Servers Console Search Call Hierarchy

No search results available. Start a search from the [search dialog...](#)

Writable Smart Insert 9:50:229

# 加上四個註釋

workspaceA - WebAppLab01/src/main/java/lab01/model/MemberBean.java

File Edit Source Refactor Navigate Search Project Run

MemberBean.java x HibernateUtils.java

```
1 package lab01.model;
2
3 import java.io.Serializable;
4
5
6
7
8
9
10
11
12
13
14
15
16 @Entity
17 @Table(name="member")
18 public class MemberBean implements Serializable {
19     private static final long serialVersionUID = 1L;
20
21     @Id
22     @GeneratedValue(strategy = GenerationType.IDENTITY)
23     private Integer id; // ObjectId
24     private String memberId; // 帳號
25     private String password; // 密碼
26     private String name; // 姓名
27     private String phone; // 電話
28     @Temporal(TemporalType.DATE)
29     private java.util.Date birthday; // 生日
30     private java.sql.Timestamp registerDate; // 會員登錄日期
31     private double weight; // 體重
32
33     public MemberBean() {
34     }
35 }
```

@Entity: 通知Hibernate，本類別中的註釋提供OR Mapping的資訊  
@Table: 說明對應的表格名稱，省略它時，預設表格名稱為類別名稱  
@Id: 說明本屬性對應的欄位為Primary Key  
@GeneratedValue(strategy=GenerationType.IDENTITY)  
說明採用資料庫提供之自動產生主鍵值做為本欄位的內容。

在這裡輸入文字來搜尋

下午 11:36  
2022/12/13



# MemberBean.java的原始碼

```
package lab01.model;

import java.io.Serializable;
import java.sql.Timestamp;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name="member")
public class MemberBean implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;           // ObjectId
    private String memberId;      // 帳號
```

1. 本類別一定要有預設建構子
2. 所有與表格欄位有對應關係的屬性，一定要編寫Getter/Setter方法

# MemberBean.java

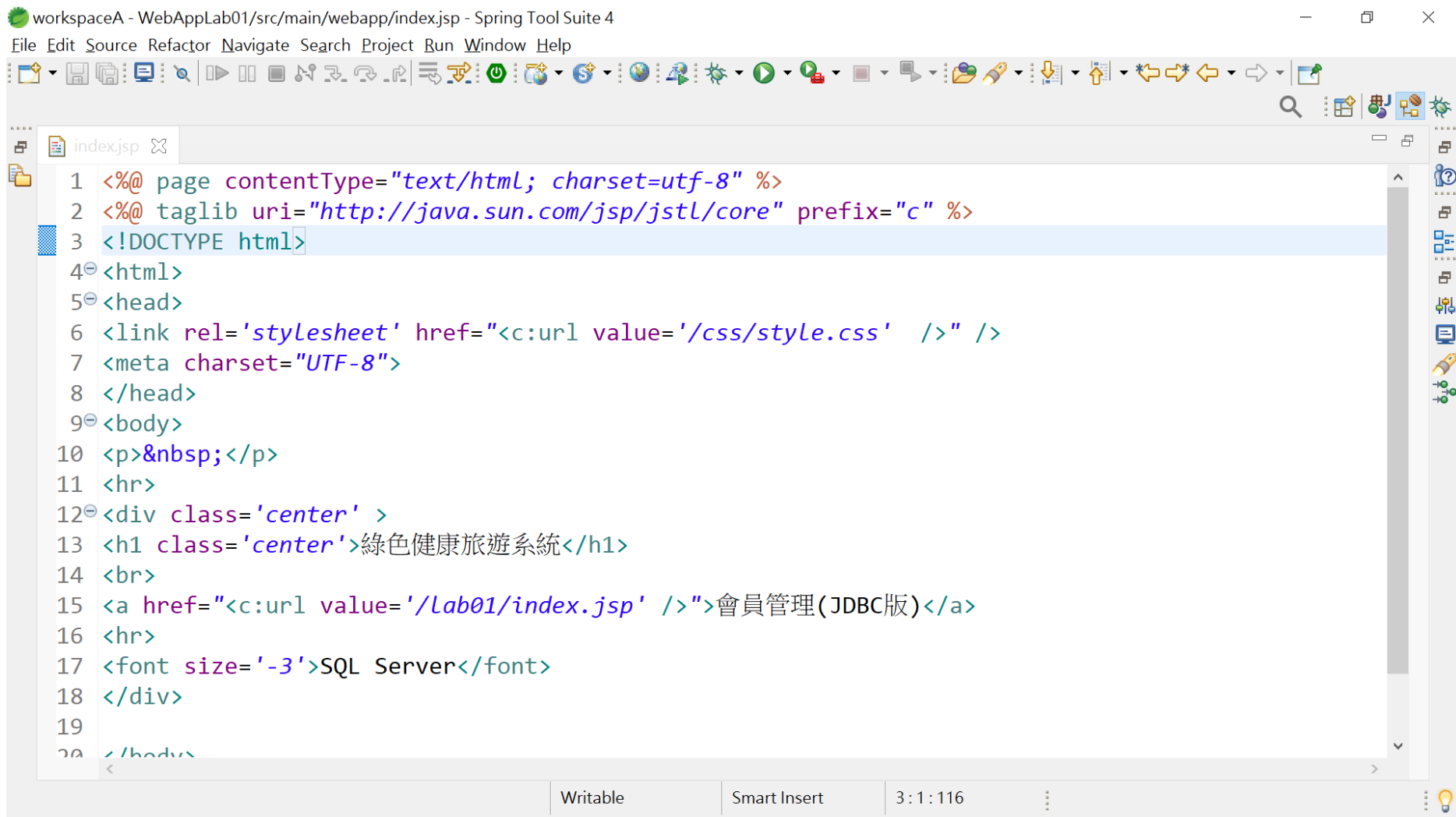
```
private String password;      // 密碼
private String name;          // 姓名
private String phone;         // 電話
@Temporal(TemporalType.DATE)
private java.util.Date birthday; // 生日
private java.sql.Timestamp registerDate; // 會員登錄日期
private double weight;        // 體重

public MemberBean() {
}

public MemberBean(String memberId, String name, String password, String phone, Date
    birthday,
    Timestamp registerDate, double weight) {
    super();
    this.memberId = memberId;
    this.name = name;
    this.password = password;
    this.phone = phone;
    this.birthday = birthday;
    this.registerDate = registerDate;
    this.weight = weight;
}
```

// 省略所有的Getter

# 修改index.jsp



workspaceA - WebAppLab01/src/main/webapp/index.jsp - Spring Tool Suite 4

File Edit Source Refactor Navigate Search Project Run Window Help

```
1 <%@ page contentType="text/html; charset=utf-8" %>
2 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <link rel='stylesheet' href="<c:url value='/css/style.css' />" /> />
7 <meta charset="UTF-8">
8 </head>
9 <body>
10 <p>&nbsp;</p>
11 <hr>
12 <div class='center' >
13 <h1 class='center'>綠色健康旅遊系統</h1>
14 <br>
15 <a href="<c:url value='/lab01/index.jsp' />">會員管理(JDBC版)</a>
16 <hr>
17 <font size='-3'>SQL Server</font>
18 </div>
19
20 </body>
```

Writable Smart Insert 3:1:116

```
1 <%@ page contentType="text/html; charset=utf-8" %>
2 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
3
4 <!DOCTYPE html>
5 <html>
6 <head>
7 <link rel='stylesheet' href="<c:url value='/css/style.css' />" /> />
8 <meta charset="UTF-8">
9 </head>
10 <body>
11 <p>&nbsp;</p>
12 <hr>
13 <div class='center' >
14 <h1 class='center'>綠色健康旅遊系統</h1>
15 <br>
16 <a href="<c:url value='/lab01/index.jsp' />">會員管理(Hibernate版)</a>
17 <hr>
18 <font size='-3'>SQL Server</font>
19 </div>
```

- 
- 修改DAO類別，用Hibernate框架提供的類別對表格內的紀錄進行增、刪、改、查等基本操作。
    - 編寫Hibernate組態檔
    - 修改相關的POJO類別，以便提供O/R Mapping的依據
    - 新建使用Hibernate技術，實作MemberDao介面的DAO類別。