



Hibernate

🕒 Last edited time	@March 19, 2025 10:12 AM
☰ Summary	(1) Session (2) Tx (3) 關聯方式

前導

📌 001 📌 Hibernate框架 (Hibernate)

- Hibernate 是一個**Java物件持久化**框架。屬於 **Object-Relational Mapping (ORM)** 技術，將 Java 物件與資料庫中的資料表進行映射。
- 它幫助 Java 開發者以面向物件的方式操作資料庫，並自動生成 SQL 查詢語句。
- Hibernate 避免了開發者直接編寫大量的 SQL 語句，提供了更高層次的抽象。
- 雖然 Hibernate 最終會生成 SQL 查詢語句來與資料庫互動，但它的核心是基於 Java 類別 (POJOs) 進行操作，底層使用 **JDBC** 與資料庫進行交互，並通過配置或註解將這些類別映射到資料庫中的表格。

📌 002 📌 持久化 (Persistence)

持久化 (Persistence) 的概念指的是將資料儲存到某個持久的存儲媒介 (如硬碟或資料庫) 中，這樣資料不會隨著程序的結束而消失。Hibernate 提供提供了簡單的 API 來操作這些資料。

📌 003 📌 JPA (Jakarta Persistence API)

JPA (Java Persistence API) 是 Java EE 中的一個標準，提供了管理資料庫操作的規範，定義了一個統一的接口，並且能夠和各種 ORM 框架 (如 Hibernate) 一起使用。

JPA 提供了一個統一的 API，將 ORM 實現與應用邏輯分離，讓開發者可以使用標準的 API 來進行資料庫操作，而不需要關注底層具體的 ORM 框架 (如 Hibernate)。

- **抽象化**：開發者不需要寫 SQL 查詢語句，直接操作 Java 物件，讓資料庫交互更簡單。
- **數據持久性**：確保資料在程序結束後仍然保存在資料庫中。
- **資料庫獨立性**：可以選擇不同的資料庫實現，並且可以在不同資料庫之間進行切換而不影響應用邏輯。

📌 004 📌 Maven依賴 (Maven Dependency)

- Maven是Java項目管理和構建工具。簡化建構過程、依賴管理和項目部署等操作。使用 `POM.xml` 來描述項目的結構和配置。
 - **約定優於配置 (Convention over Configuration)**：採用專案常見的預設結構，減少人為配置，降低了學習成本。
 - `src/main/java` → 存放 Java 原始碼、`src/main/resources` → 存放資源文件 (如 `application.properties`)、`src/test/java` → 存放測試代碼、`target/` → Maven 編譯後的輸出目錄
 - **依賴關係管理 (Dependency Management)**：如何處理和管理應用程序所依賴的第三方庫、框架和其他模組。(1)自動下載所需的 **JAR 檔案**(2)解決**依賴衝突**(3)版本管理
- ▼ **常見命令**：`mvn clean` → 清除 `target/` 目錄、`mvn compile` → 編譯專案、`mvn test` → 執行測試、`mvn package` → 打包成 JAR 或 WAR 檔案、`mvn install` → 將專案安裝到本地倉庫、`mvn dependency:tree` → 顯示依賴樹，分析依賴衝突

```
<dependencies>
<!-- Spring 依賴 -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>5.3.10</version>
</dependency>
```

```

<!-- JUnit 依賴 -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.1</version>
  <scope>test</scope>
</dependency>
</dependencies>

```

Maven 依賴作用域 (Dependency Scope) 詳細解釋

作用域	是否在編譯時可用	是否在測試時可用	是否在運行時可用	是否包含在最終打包	屬性名稱 與應用	說明
compile	✓ 是	✓ 是	✓ 是	✓ 是	groupId 預設作用域，適用於 <code>org.springframework</code> 等	組 ID，通常代表 組織或公司
test	✗ 否	✓ 是	✗ 否	✗ 否	artifactId JUnit, Mockito	庫名稱，代表 具體的模組或專案名稱 (如 <code>spring-core</code>)
provided	✓ 是	✓ 是	✗ 否	✗ 否	version Servlet API (Tomcat 提供)	版本號，指定 所需的庫版本 (如 <code>5.3.20</code>)
runtime	✗ 否	✓ 是	✓ 是	✓ 是	scope (可選)	作用域，控制 依賴的可用範圍 (如 <code>compile</code> , <code>test</code>)
system	✓ 是	✓ 是	✓ 是	✓ 是		只在運行時需要，如 JDBC 驅動
import	✗ 否	✗ 否	✗ 否	✗ 否		需要指定本地 JAR 路徑，不推薦使用
						用於 <code><dependencyManagement></code> ，引入 BOM

Maven 依賴的主要屬性

特性	依賴 (Dependency)	插件 (Plugin)
作用	項目運行和編譯所需的外部庫或 JAR 文件	執行構建過程中的各種任務，如編譯、測試、打包、部署等
用途	提供代碼庫，支持應用程式的運行或測試	控制項目的構建流程，執行具體的構建任務
加載方式	透過 <code><dependencies></code> 標籤指定，並自動下載相關 JAR	透過 <code><plugins></code> 標籤指定，並在 Maven 執行過程中觸發
配置位置	配置於 <code><dependencies></code> 部分	配置於 <code><build><plugins></code> 部分
示例	<code>spring-core</code> 、 <code>JUnit</code> 、 <code>Jackson</code> 等外部庫	<code>maven-compiler-plugin</code> 、 <code>maven-surefire-plugin</code> 、 <code>maven-jar-plugin</code>
生命週期	依賴會被加載到 classpath，在編譯、測試和運行時使用	插件會在構建過程中執行特定任務，如編譯、打包或部署

📌 005 📌 連接資訊 (Connection Information)

▼ 連線資訊：Hibernate 需要 **JDBC 驅動** 才能與資料庫連線，這部分透過 `hibernate.connection.*` 屬性設定：**指** `driver_class`、`url`、`username`、`password`

屬性	用途	範例
<code>hibernate.connection.driver_class</code>	指定 JDBC 驅動類別	<code>com.mysql.cj.jdbc.Driver</code>
<code>hibernate.connection.url</code>	指定資料庫 URL	<code>jdbc:mysql://localhost:3306/mydb?</code>
<code>hibernate.connection.username</code>	資料庫登入帳號	<code>root</code>
<code>hibernate.connection.password</code>	資料庫登入密碼	<code>password</code>

```

Properties properties = new Properties();
properties.put("hibernate.connection.driver_class", "com.mysql.cj.jdbc.Driver");
properties.put("hibernate.connection.url", "jdbc:mysql://localhost:3306/mydb?useSSL=false");
properties.put("hibernate.connection.username", "root");
properties.put("hibernate.connection.password", "password");
configuration.setProperties(properties);

```

- 映射資訊：`Configuration.addAnnotatedClass(Entity .class)`：將類別標註為永續類別。
- 進階資訊
 - DIALECT：將hibernateAPI轉換成哪一種廠牌的敘述**指** `org.hibernate.dialect.SQLServerDialect`、`org.hibernate.dialect.MySQL8Dialect`

▼ Hibernate Schema Management : `hbm2ddl.auto` 是用於控制 Hibernate 如何處理資料庫的結構生成**指**(開發) `update`、(生產) `validate`、(測試) `create-drop`

選項	用途	適用環境	影響
<code>update</code>	自動更新表結構，不刪除現有資料	開發環境	⚠ 可能發生資料庫結構不一致
<code>validate</code>	驗證表結構是否正確，不修改，僅拋出異常	生產環境	❌ 不會自動更新結構
<code>create</code>	每次重新創建表格 (清空資料)	開發、測試環境	⚠ 刪除所有資料
<code>create-drop</code>	應用啟動時創建表格，關閉時刪除表格	測試環境	🔪 測試結束後不留痕跡
<code>none</code>	完全不進行任何 Schema 變更	生產環境	✅ 避免意外改動

核心

📌 006 📌 會話工廠 (SessionFactory)

`SessionFactory` 是 Hibernate 中用來創建和管理 `Session` 的工廠類。它是整個 Hibernate 應用程式的核心對象之一，用來配置和初始化 Hibernate 的數據庫連接、映射設置以及其他重要的設置。它是多線程安全的，並且通常是應用程式中唯一一個 `SessionFactory` 實例，耗費資源所以一個專案應該只建立一個。

▼ 手動設定SessionFactory：先new一個configuration物件和properties物件。properties物件的put方法將資訊放進去。configuration.setProperty(property)來完成設定。buildSessionFactory產生工廠

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

import java.util.Properties;

public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
            // 1. 創建 Configuration 物件
            Configuration configuration = new Configuration();

            // 2. 創建 Properties 物件
            Properties properties = new Properties();
            properties.put("hibernate.connection.driver_class", "com.mysql.cj.jdbc.Driver"); // JDBC 驅動
            properties.put("hibernate.connection.url", "jdbc:mysql://localhost:3306/mydb?useSSL=false"); // 資料庫 URL
            properties.put("hibernate.connection.username", "root"); // 資料庫帳號
            properties.put("hibernate.connection.password", "password"); // 資料庫密碼
            properties.put("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect"); // Hibernate 方言
            properties.put("hibernate.show_sql", "true"); // 顯示 SQL
            properties.put("hibernate.hbm2ddl.auto", "update"); // 自動更新 Schema

            // 3. 設定 Properties 到 Configuration
            configuration.setProperties(properties);

            // 4. 加載 Annotated Class
            configuration.addAnnotatedClass(Student.class);

            // 5. 建立 SessionFactory
            sessionFactory = configuration.buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError("初始化 Hibernate 失敗: " + ex);
        }
    }
}
```

```

    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}

```

- **配置和初始化：** `SessionFactory` 需要在應用啟動時進行初始化，它加載 Hibernate 配置文件（如 `hibernate.cfg.xml`）和映射文件（如 `.hbm.xml` 或註解）。
- **一次性創建：** 通常在應用啟動時創建一次，並在應用生命週期內持久保存。在應用運行期間，`SessionFactory` 不會經常變動。
- **線程安全：** `SessionFactory` 是線程安全的，因此可以在多個線程中共享。
- **創建 Session：** `SessionFactory` 主要作用是為每個數據庫交互創建一個 `Session`。

工作原理：

- 它會根據配置文件初始化 Hibernate，設置與數據庫的連接信息、緩存、事務等配置，並提供創建 `Session` 的方法。
- 當你需要與數據庫進行交互時，`SessionFactory` 會創建一個 `Session` 實例。
- `SessionFactory` 介面：類別是 `ThreadSafe`，允許多個執行緒共用一個 `SessionFactory` 物件。建立時需要提供組態資訊。耗費資源，每個應用系統應只建一個 `SessionFactory` 物件。
- `session` 介面：不是 `ThreadSafe`，一個 `session` 物件只能給一個執行緒使用。`session` 型別的變數必須是區域變數，且使用時才開啟，用完立刻關閉。開啟交易時才能對表格進行增刪改查。

007 Session（會話）

`Session` 是 Hibernate 與資料庫進行交互的主要接口，通過它來進行資料庫的操作（增、刪、改、查）。它提供了一系列方法來操作資料（例如：保存、更新、查詢等）。

- `save()`：將一個物件持久化到資料庫中（即插入資料）。
- `update()`：更新一個已經存在的資料庫紀錄。
- `delete()`：刪除資料庫中的紀錄。`get()` / `load()`：根據主鍵查詢資料。
- `createQuery()`：用來創建 HQL (Hibernate Query Language) 查詢。

- **事務管理：** `Session` 用於管理事務（即開始、提交或回滾事務）。

- `beginTransaction()`：開啟一個新的資料庫事務，Hibernate 會建立一個 `transaction` 物件，SQL 操作不會立即執行，直到 `commit()`

- ▼ `getTransaction()`：用來取得當前的 `Transaction` 物件，可以用來確認事務的狀態 `isActive()` 不會新建事務，只是獲取當前的 `Transaction` 物件

```
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
```

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

```
// 執行 CRUD 操作
session.save(newObject);
session.get(YourEntity.class, id);
```

```
// 提交事務
```

`Session` 的生命週期：每個 `Session` 代表一次操作，通常在一個事務中使用。

- 一旦完成操作後，`Session` 會自動關閉。它是一個短期存在的物件，通常在一次操作後就不再需要，並且會釋放資源，在請求結束時關閉。Ⓜ防止內存洩漏或資料庫連線問題。
- `Session` 是非線程安全，將其範圍限制在一個事務或操作內，確保每個線程都擁有自己的 `Session` 實例。
- 關閉 `session` 物件不等於關閉資料庫連線。如果程式是在提供 `connectionpool` 的環境下執行 `close()`，會將 `connection` 物件還回連接池而不會關閉它。

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

Hibernate Session 的工作流程：

1. **創建 SessionFactory**：應用啟動時創建一個 `SessionFactory` 實例。
 2. **開啟 Session**：通過 `SessionFactory` 獲得 `Session` 實例，開啟一個新的資料庫會話。
 3. **執行操作**：使用 `Session` 來執行資料的增、刪、改、查等操作。
 4. **提交事務**：對於需要持久化操作的資料變更，需要進行事務提交。
 5. **關閉 Session**：操作完成後，`Session` 會被關閉，釋放資源。
- 使用了 **try-with-resources** 來自動關閉 `SessionFactory` 和 `Session`。這樣當程式執行完成後，這些資源會自動關閉，而不需要手動調用 `close()` 方法。

```
try (SessionFactory factory = new Configuration().configure("hibernate.c
    .addAnnotatedClass(Student.class).buildSessionFactory();
    Session session = factory.getCurrentSession()) {

    session.beginTransaction();

    // 儲存學生資料
    Student student = new Student("John", "Doe", "john.doe@example.com
    session.save(student);

    session.getTransaction().commit();
}
```

特性	<code>openSession()</code>	<code>getCurrentSession()</code>
返回的 Session	返回一個新的、獨立的 <code>Session</code> 物件。每次呼叫都會創建一個新的會話。	返回當前的 <code>Session</code> 物件，如果沒有可用的會話，則創建一個新的會話。
事務管理	必須手動管理事務， <code>openSession()</code> 不會自動綁定事務。	<code>getCurrentSession()</code> 會自動綁定事務。當程式交易跨過多個方法時，每個方法都一定要以 <code>getCurrentSession</code> 取得 <code>session</code> 物件。
Session 的生命週期	必須顯式開啟和關閉，適合於短期的操作。	會自動管理 <code>Session</code> 的生命週期（當前事務結束時自動關閉）。無需 <code>session.beginTransaction()</code>
事務處理	在進行事務處理時，使用 <code>beginTransaction()</code> 開始事務並提交。	通常與 Spring 的事務管理協作，事務處理會自動與當前事務綁定。
用途	用於獨立手動管理 <code>Session</code> 的情境，對需要精細控制事務的情況很有用。	用於基於事務的操作，通常在 Spring 等框架中與事務管理自動集成。
性能考量	每次呼叫都會創建新的 <code>Session</code> ，性能開銷較高。	會在同一個事務範圍內重用當前的 <code>Session</code> ，能夠減少資源的浪費。
關閉 Session	開啟的 <code>Session</code> 必須手動關閉，否則會造成資源浪費。	當事務結束時， <code>Session</code> 會自動關閉，無需顯式關閉。無需 <code>session.close()</code>
線程安全	<code>Session</code> 是非線程安全的，並且它的生命週期是由開發者來手動控制的。	<code>Session</code> 是由事務管理器（如 Spring）來管理，因此能保證線程安全。
組態設定	不需要做任何組態設定就可以使用 <code>openSession</code> 方法。	必須在組態設定加入 <code>current_session_context_class</code> 。
跨方法	不能跨過多個 DAO 類別進行交易	必須使用此方法進行跨 DAO 類別進行交易。

📌 008 📌 Hibernate 物件生命週期（Object Lifecycle）

狀態	描述	OID 是否存在	Session 關聯	影響資料庫	如何轉換到其他狀態	
Transient（暫時狀態）	剛 <code>new</code> 出來的物件，尚未儲存到資料庫	❌ 無 OID	❌ 無	❌ 不影響	<code>session.persist(obj)</code> → Persistent	<code>session.save(obj);</code> <code>session.saveOrUpdate(obj)</code>

Persistent (永續狀態)	受 Session 管理，變更會自動同步到資料庫	✓ 有 OID	✓ 受管理	✓ 影響 (變更會自動 flush)	<code>session.detach(obj)</code> → Detached / <code>session.remove(obj)</code> → Removed	<code>session.get()</code> 取出
Detached (脫管狀態)	來自資料庫的物件，但 <code>session.close()</code> 了	✓ 有 OID	✗ 無	✗ 不影響	<code>session.merge(obj)</code> → Persistent	<code>session.detach(obj); session.c</code>
Removed (刪除狀態)	<code>session.remove(obj)</code> 後，待 <code>commit</code> 刪除	✓ 有 OID	✓ 受管理	✓ 影響 (<code>commit()</code> 會刪除)	<code>session.persist(obj)</code> → Persistent	

📌 009 📌 Session 方法 (Session methods)

`session.persist(obj)`

▼ 若主鍵 (PK) 為自增 (`@GeneratedValue(strategy = GenerationType.IDENTITY)`)，Hibernate 會立刻執行 `INSERT INTO`，讓資料庫生成主鍵值。

```
@Entity
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // 自增主鍵
    private Long id;
    private String name;
}
```

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

```
Member member = new Member();
member.setName("John Doe");
```

```
session.persist(member); // **立即發出 INSERT**
tx.commit();
```

```
INSERT INTO Member (name) VALUES ('John Doe');
```

▼ 若 PK 由程式設定 (手動指定 `setId(123L)`)，Hibernate 不會馬上 `INSERT`，而是等到 `commit()` 或 `flush()` 時才寫入。

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

```
Member member = new Member();
member.setId(123L); // 自行提供主鍵
member.setName("Jane Doe");
```

```
session.persist(member); // **不立即發送 INSERT**
tx.commit(); // **這時候才 INSERT**
```

`session.remove(obj)`

▼ 只能刪除 **Persistent (永續)** 狀態的物件。若傳入 **Transient (臨時)** 物件 (沒有 OID)，Hibernate 會拋出例外。 `session.commit()` 之後，物件才真正從資料庫刪除。

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

```
Member member = session.get(Member.class, 1L); // 從 DB 取出 Persistent 物件
session.remove(member); // **標記為刪除**
tx.commit(); // **這時候才執行 DELETE**
```

```
DELETE FROM Member WHERE id = 1;
```

```
//錯誤情況：刪除 Transient 物件
Member member = new Member(); // Transient, 沒有 OID
session.remove(member); // ❌ Hibernate 會拋出錯誤
```

```
org.hibernate.TransientObjectException: The given object has a null identifier.
```

`session.merge(obj)`

▼ 傳入的參數物件不會變成 **Persistent**，Hibernate 會回傳新的 **Persistent 物件**。若有 **OID**，則發送 **UPDATE**，否則發送 **INSERT**。適合用來更新 Detached（脫管）物件。

```
Session session1 = sessionFactory.openSession();
Member detachedMember = session1.get(Member.class, 1L);
session1.close(); // **物件變成 Detached**

Session session2 = sessionFactory.openSession();
Transaction tx = session2.beginTransaction();

Member updatedMember = session2.merge(detachedMember); // **回傳 Persistent 物件**
updatedMember.setName("Updated Name"); // **這個物件受 Hibernate 管理**

tx.commit();

UPDATE Member SET name = 'Updated Name' WHERE id = 1;
```

`session.refresh(obj)`

▼ 重新從資料庫加載物件，覆蓋 Hibernate 快取 中的物件狀態。適合當別人改變了資料庫的值時，讓 Hibernate 重新取得最新值。

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

Member member = session.get(Member.class, 1L);
System.out.println(member.getName()); // 可能是 "Old Name"

// **假設這時候資料庫的 `name` 欄位被別人改成 "New Name"**
session.refresh(member);

System.out.println(member.getName()); // 這時候會是 "New Name"

tx.commit();
```

方法	狀態變化	何時執行 SQL	主要用途
<code>persist(obj)</code>	Transient → Persistent	立即 INSERT （若 PK 為自增）或 commit 時 INSERT	新增資料
<code>remove(obj)</code>	Persistent → Removed	commit() 時 DELETE	刪除資料
<code>merge(obj)</code>	Detached → Persistent（回傳值）	UPDATE （若 OID 存在），否則 INSERT	更新 Detached 物件

<code>refresh(obj)</code>	Persistent (重載最新值)	<code>SELECT</code> 重新查詢	同步資料庫最新狀態
---------------------------	--------------------	--------------------------	-----------

方法	狀態變化	是否影響傳入物件	何時執行 SQL	主要用途	需不需要 OID
<code>save(obj)</code>	Transient → Persistent	✓ 傳入的物件變成 Persistent	立即執行 <code>INSERT</code>	新增資料	✗ 不需要 (會自動生成)
<code>update(obj)</code>	Detached → Persistent	✓ 傳入的物件變成 Persistent	<code>commit</code> 或 <code>flush</code> 時 <code>UPDATE</code>	更新 Detached 物件	✓ 需要 OID
<code>merge(obj)</code>	Detached → Persistent (回傳值)	✗ 傳入的物件還是 Detached, 但回傳值是 Persistent	<code>UPDATE</code> (若 OID 存在), 否則 <code>INSERT</code>	更新 Detached 物件, 但不影響原物件	✓ 需要 OID (沒 OID 則 <code>INSERT</code>)

📌 010 📌 Session Caching 機制 (Caching)

在 Hibernate 中, Session 是與資料庫互動的核心物件, 它提供了 **第一級快取 (L1 Cache)**, 讓系統在單一交易內避免不必要的查詢與更新。

1. 第一級快取 (L1 Cache)

- 作用範圍：單一 Session。調用 `session.get()` 或 `session.find()` 讀取某筆資料時

▼ (1) 先檢查 Session 緩存內是否已經有該物件 (根據 **OID**)。 (2) 若已存在, 則直接從快取中取得 (避免發送 SQL 查詢)。 (3) 如果不存在, 才會發送 `SELECT` 查詢資料庫, 然後將查詢結果放入緩存。

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

Member member1 = session.get(Member.class, 1L); // 這裡會執行 SELECT
Member member2 = session.get(Member.class, 1L); // 這裡不會執行 SQL, 因為已經在 L1 Cache

tx.commit();
session.close(); // Session 關閉後, L1 Cache 也會被清除
```

- ▼ `tx.commit()`：先自動 `flush()`, 然後真正提交交易。 `session.flush()` 將緩存中的變更強制寫入資料庫, 但不會提交交易。

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

Member member = session.get(Member.class, 1L);
member.setName("Updated Name"); // 這裡只會更新 L1 Cache, 不會立即執行 UPDATE

tx.commit(); // Hibernate 自動 flush(), 然後 commit()

session.close();
```

- ▼ 清除快取：`session.clear()`：清空整個 L1 Cache, 但不會影響資料庫。; `session.evict(obj)`：清除特定物件的 L1 Cache。'

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

Member member = session.get(Member.class, 1L);
session.evict(member); // 這樣下一次讀取 member 會重新查詢資料庫

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


4. 第二級快取 (L2 Cache)

- 範圍：跨 Session，但限定於相同的 SessionFactory，必須額外設定（L1 Cache 是內建的）

▼ 常見的 L2 Cache 實作：

(1)Ehcache(2)Hazelcas(3)tRedis（透過 Hibernate ORM 5+ 外掛）

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}
```

- 主要用於：讀取較頻繁但變更較少的資料（例如產品清單、會員等）。

特性	第一級快取 (L1)	第二級快取 (L2)
範圍	單一 Session	跨 Session (SessionFactory 內共享)
內建支援	內建（無需額外設定）	需要額外設定
清除時機	session.close() 時清除	直到資料更新或手動清除
影響範圍	只影響當前交易	多個交易間共享資料

📌 011 📌 事務的四個基本特性 (ACID)

事務 (Transaction) 是指在數據庫管理系統 (DBMS) 中，對一組操作（如插入、更新、刪除等）進行的一個完整的、不可分割的操作單位。

事務管理 (Transaction Management) 是對數據庫操作進行控制，以確保一組操作（通常是多個操作或多個數據庫查詢）要麼完全成功，要麼完全回滾，從而保證數據的一致性、可靠性和完整性。

位置：交易應該由 **Service 層負責管理**，**DAO 層不應該開啟或關閉交易**。錯誤處理：在 `try` 區塊內開啟交易，若發生異常則 `catch` 區塊執行 `tx.rollback()`。

1. **原子性 (Atomicity)**：事務中的所有操作要麼完全成功或失敗。事務中包含的所有操作會被視為一個不可分割的單元。如果其中某個操作失敗，那麼整個事務應該回滾，所有已經執行的操作都應撤銷。
 - **指**銀行轉帳，從帳戶A轉錢到帳戶B，如果扣款成功但存款失敗，那麼整個轉帳事務應該回滾。
2. **一致性 (Consistency)**：在事務開始和結束時，數據庫都處於一致的狀態。事務的執行不應該違背數據庫的約束和規則，並且事務執行後，數據庫必須轉移到一個合法的狀態。
 - **指**如果事務涉及將錢從一個銀行帳戶轉到另一個帳戶，那麼最終結果應該是帳戶A的餘額減少，帳戶B的餘額增加，而不是兩個帳戶的餘額總和變化。
3. **隔離性 (Isolation)**：確保同時執行的事務彼此之間不會相互干擾。不同的事務操作應該看不到彼此的中間狀態。具體來說，這意味著即使多個事務同時進行，也不會造成數據不一致。
 - **指**在兩個事務同時執行的情況下，如果兩個事務都在更新同一條數據，則應保證其中一個事務的修改不會被另一個事務“看到”直到它提交。
4. **持久性 (Durability)**：一旦事務提交，對數據的修改應該是永久性的，即使系統崩潰或出現其他故障，數據仍然能夠保證不會丟失。
 - **指**轉帳事務提交後，無論系統是否發生崩潰，兩個帳戶的更新都會被永久保存。

📌 012 📌 `current_session_context_class` (Annotation)

- ▼ `current_session_context_class` 每個選項都指定了 Hibernate 如何管理當前會話的上下文，用來控制 Hibernate 如何管理當前的會話 (Session)。

▼ **thread**（默認設置）Hibernate 會將 **Session** 綁定到當前線程（Thread）。每個線程（Thread）會有一個獨立的 **Session**，並且當線程結束時，會話也會結束。

- **應用場景**：適用於線程安全的環境，例如單獨的應用程序中，每個線程都使用不同的 **Session** 進行數據操作。

- **配置**：`<property name="hibernate.current_session_context_class">thread</property>`

- **如何工作**：每個線程都有自己的 **Session** 實例，並且 Hibernate 會在每個線程中維護當前的 **Session**。這是最常見的設置，適用於大多數環境。

▼ **jta**：Hibernate 會將 **Session** 綁定到 **Java Transaction API (JTA)** 的事務上下文中。這通常用於分布式事務環境，其中多個資源（如數據庫、消息隊列等）共享一個全局事務。

- **應用場景**：適用於使用 JTA 事務管理器的應用程序，特別是分布式系統或企業應用（如 EJB 或 J2EE 應用）中。

- **如何工作**：在這種模式下，Hibernate 不再直接綁定 **Session** 到線程，而是依賴於 JTA 事務上下文來管理 **Session**。通常需要在容器（如 JBoss、WebLogic）中配置 JTA 事務管理器。

▼ **managed**：這個模式是設置 **Session** 綁定到容器管理的事務上下文中。在容器管理的環境中（例如使用 Spring 的事務管理），**Session** 被綁定到當前的事務上下文。

- **應用場景**：適用於 Spring 或其他基於容器的應用，在這些應用中，容器會自動管理事務和 **Session** 的生命週期

- **如何工作**：Hibernate 會根據容器管理的事務來決定當前的 **Session**。這通常與 Spring 或 Java EE 的容器一起使用，其中 Spring 管理事務，並且 **Session** 被綁定到 Spring 事務管理中。

- **線程綁定模式（thread）**：線程本地存儲(ThreadLocal)來跟蹤每個線程的當前會話。每個線程都有自己專屬的 **Session**，不同線程不會共享同個 **Session** 實例。簡單且高效，特別適用於多線程環境。

- **事務綁定模式（jta）**：使用 J2EE 或 Spring 事務管理，則 **Session** 的綁定是基於事務的，而非線程。該模式允許在分布式事務中正確地管理 **Session**，適用於需要跨多個資源進行事務控制的應用。

- **容器管理模式（managed）**：這種模式依賴於應用容器的事務管理。在 Spring 或 Java EE 應用中，Hibernate 不直接管理會話，而是由容器的事務管理器來處理會話的綁定與生命週期。

選擇合適的 `current_session_context_class`

- 如果您的應用是基於 **單線程** 或 **簡單的桌面應用程序**，通常會選擇 **thread** 模式，這是默認的設置。
- 如果您的應用需要支持 **分布式事務** 或使用 **JTA** 進行事務管理（如在企業應用中），則應選擇 **jta**。
- 如果您使用 **Spring** 來管理事務，並希望 Hibernate 的 **Session** 與 Spring 事務一致，則應選擇 **managed**。

📌 013 📌 連接池（Connection Pool）

連接池（Connection Pool）是一種資源管理機制，用於管理資料庫連線的重複使用。其主要目的是減少因為每次請求都建立新資料庫連線所帶來的資源消耗和性能損失。當多個應用程式請求資料庫連線時，連接池會提供已經建立好的連線，而不是每次都重新建立新連線。這樣可以提高應用程式性能，減少延遲。建立連線物件很耗費資源所以不再使用連線時不應銷毀而應該放在連線池內。

▼ 連接池的工作原理：（1）初始化連接池（2）取得連線（3）使用連線（4）釋放與銷毀

1. **初始化連接池**：當應用程式啟動時，連接池會預先建立一定數量的資料庫連線並存放在池中。這些連線保持打開狀態，準備好供應用程式使用。
2. **取得連線**：當程式需要資料庫連線時，它會向連接池請求一個連線。連接池會檢查是否有可用的連線。如果有，則返回一個連線；如果沒有，則根據配置創建新的連線，並將其提供給應用程式使用。
3. **使用連線**：當應用程式使用完資料庫連線後，並不會直接關閉該連線，而是將其歸還給連接池，以便其他請求可以重複使用這個連線。這樣避免了不斷創建和銷毀連線的資源浪費。
4. **釋放與銷毀**：當應用程式不再需要某個連線時，連線會被釋放回池中。在某些情況下，當連線空閒超過設定的時間，或者達到最大空閒數量時，連接池會自動銷毀這些連線。

▼ 連接池的優點：（1）提高效能（2）資源共享（3）減少資源消耗（4）簡化管理

1. **提高效能**：減少了每次請求都建立新連線的開銷，尤其是在高並發的情況下，可以大幅度提高應用程式的性能。
2. **資源共享**：多個請求可以共享連接池中的有限連線資源，避免了資源的浪費。
3. **減少資源消耗**：建立連線是昂貴的操作，重複使用已經存在的連線可以減少系統資源消耗。

4. **簡化管理**：大多數連接池提供了自動檢查和維護連線的功能，會自動重用或替換失效的連線，減少了程式的複雜性。

▼ 連接池配置參數：(1) **最大連線數量** (2) **最小連線數量** (3) **最大空閒時間** (4) **連接超時時間** (5) **檢查間隔**

1. **最大連線數量** (Max Connections)：連接池中最多允許的連線數量。這個數值一般是根據資料庫的性能和應用程式的需求來設定的。
2. **最小連線數量** (Min Connections)：連接池中最少保留的連線數量。在系統啟動時會預先創建的連線數量。
3. **最大空閒時間** (Max Idle Time)：一個連線可以在池中保持空閒的最大時間，超過這個時間後，池中的該連線會被銷毀。
4. **連接超時時間** (Connection Timeout)：當請求連線時，若連接池中的連線數量已達上限，等待連線的最大時間。超過此時間將拋出異常。
5. **檢查間隔** (Validation Interval)：連接池檢查和移除失效連線的頻率。

- Java 連接池框架：(1) **HikariCP**：輕量級且高效能的 Java 連接池，適用於高併發環境。(2) **Apache DBCP**：經典且穩定的 Java 連接池框架。(3) **C3PO**：功能豐富的 Java 連接池，但性能相對較低。

特徵	連接池	直接連接
性能	高效，通過重用連線避免頻繁創建和銷毀連線	每次請求都需要建立新的連線，性能較低
資源消耗	減少了建立和銷毀連線的資源消耗	每次都需要新建連線，消耗較多資源
可擴展性	支援高並發，適合大型應用	隨著請求數量增加，可能會面臨效能瓶頸
配置靈活性	可以配置最大連線數量、最大空閒時間等多種參數	無法配置連線池，較為簡單
錯誤處理	連接池提供自動檢查失效連線並替換功能	如果連線失效，需要手動處理錯誤

實體

📌 014 📌 POJO類別 (Plain Old Java Object, JPA Entity)

- 不能繼承特定類別或實作額外的介面（除了 `Serializable`）。
- 主要負責資料儲存，不負責商業邏輯
- 必須有**預設建構子**（否則 Hibernate 會拋出錯誤）。
- 屬性類型應該使用**包裝類型**（例如 `Integer` 而不是 `int`），這樣可以處理 `null` 值。

▼ 屬性應該是 `private`，使用 `getter/setter` 來存取

```
public class Address implements Serializable {
    private static final long serialVersionUID = 1L;

    private String city;
    private String street;

    public Address() {} // 預設建構子

    public Address(String city, String street) {
        this.city = city;
        this.street = street;
    }

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }

    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }
}
```

📌 015 📌 Persistent 類別 (JPA Entity)

- 不能是 `final` 類別 (Hibernate 需要產生代理類，來執行延遲加載)，是 Pojo 類別的擴展，主要負責映射。

▼ 必須標註 `@Entity`，並且至少有一個 `@Id` 作為主鍵

```
@Entity
@Table(name = "members")
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    // 必須要有預設建構子
    public Member() {}

    public Member(String name) {
        this.name = name;
    }

    // Getter 和 Setter
}
```

概念	說明	範例
<code>@Lob</code>	(Large Object)用來存儲大量資料，可是二進制資料（圖片、音頻、視頻）或文字資料（長文本、文件）	<code>@Lob</code> 註解用來標註屬性，對應資料庫中的大物件類型。指 <code>byte[]</code> <code>fileData</code> 或 <code>String textData</code>
<code>@Entity</code>	在 Hibernate 中， <code>@Entity</code> 註解標註的類必須具有唯一名稱，避免映射衝突。	指例如 <code>Student</code> 和 <code>Teacher</code> 是合法的，但兩個 <code>Student</code> 類名會衝突。
<code>@Table</code>	可以在 <code>@Table</code> 註解中設置唯一約束，用來保證資料表中的某些欄位唯一。	<code>@Table(name = "student", uniqueConstraints = {@UniqueConstraint(columnNames = {"email"})})</code>
<code>@Id</code>	<code>@Id</code> 註解標註的主鍵欄位可以是基本資料型別或包裝類別。包裝類別支持 <code>null</code> 值	<code>@Id @GeneratedValue(strategy = GenerationType.IDENTITY) private Long id;</code>
<code>@transient</code>	未使用 <code>static</code> 和 <code>transient</code> 的實例變數都會有對應表格中的欄位。	靜態變數並不是在物件裡面，而是所以實例共用一份靜態變數，因此不會產生對應的欄位。

016 PK設計 (Primary Key)

- 不要使用業務相關的欄位當 PK（例如：身分證字號、電子郵件等）。
- 使用數字型 ID（`Long`、`UUID`）來做主鍵，並讓資料庫自動生成
- ▼ 如果需要全域唯一識別，可以使用 `UUID`

```
@Id
@GeneratedValue(generator = "UUID")
@GenericGenerator(name = "UUID", strategy = "org.hibernate.id.UUIDGenerator")
private String id;
```

▼ Hibernate 會先儲存沒有外鍵的紀錄，再儲存有外鍵的紀錄：FK依賴PK，如果先插入有外鍵的資料，但外鍵參照的資料還沒插入，就會導致外鍵約束錯誤 (Foreign Key Constraint Violation)。

```
@Entity
public class Parent {
    @Id @GeneratedValue
    private Long id;
```

```

    @OneToMany(mappedBy = "parent", cascade = CascadeType.ALL)
    private List<Child> children;
}

@Entity
public class Child {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "parent_id") // 外鍵指向 Parent
    private Parent parent;
}

INSERT INTO Parent (id) VALUES (1); -- 先插入父表（無外鍵）
INSERT INTO Child (id, parent_id) VALUES (100, 1); -- 再插入子表（有外鍵）

```

📌 017 📌 映射方式（Mapping）

▼ 映射方式：（1）**Field Access**：寫在實例變數之前，推薦此種方法（2）**Property Access**：寫在getter方法之前。注解（`@Column`、`@Id` 等）應用於實體類別的字段。只能二選一，不能混用。

```

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

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

    // getter 和 setter 方法
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

📌 018 📌 `@GeneratedValue` 註釋（`@GeneratedValue`）

策略名稱	說明	相關特性 - 新增物件時 <code>oid</code> 必須為 <code>NULL</code> 。	適用場景
------	----	---	------

IDENTITY	使用資料庫自動生成的 ID（通常是自增的整數）。	<code>oid</code> 欄位必須是整數。	較適合單一平台或本地數據庫，特別是在自增欄位使用的情況。
AUTO	讓 Hibernate 自動選擇適合的策略，根據資料庫的特性選擇。	<code>oid</code> 欄位必須是整數。	適用於簡單應用，Hibernate 自動選擇策略，適用於大多數資料庫。
UUID	使用全球唯一識別碼（UUID）作為主鍵。通常是字符串形式。	<code>oid</code> 欄位必須是 <code>String</code> 類型。	較適合跨平台應用，尤其是需要全球唯一識別碼的分散式系統。性能較差，因為生成 UUID 較為耗時。

📌 019 📌 `@Temporal` 註釋（`@Temporal`）

`@Temporal` 用於將 `java.util.Date` 或 `java.util.Calendar` 類型的欄位映射到資料庫中的日期和時間型別。可指定存儲的精度，即將其映射為 **日期**（`DATE`）、**時間**（`TIME`）或 **日期時間**（`DATETIME`）。

<code>@Temporal(TemporalType.DATE)</code>	將 <code>java.util.Date</code> 對象映射為 只包含日期 的資料庫欄位（不包含時間）。這會對應到 SQL 資料庫中的 <code>DATE</code> 類型。
<code>@Temporal(TemporalType.TIME)</code>	將 <code>java.util.Date</code> 對象映射為 只包含時間 的資料庫欄位（不包含日期）。這會對應到 SQL 資料庫中的 <code>TIME</code> 類型。
<code>@Temporal(TemporalType.TIMESTAMP)</code>	將 <code>java.util.Date</code> 對象映射為 日期與時間 的資料庫欄位。這會對應到 SQL 資料庫中的 <code>DATETIME</code> 或 <code>TIMESTAMP</code> 類型。

📌 020 📌 `@JoinColumn` 與 `mappedBy` 的關係解析

Hibernate 中，**外鍵 (Foreign Key, FK)** 通常設定在 "**多方 (Many Side)**"，而 `mappedBy` 則是在 "**擁有關係的一方 (Owning Side)**" 的相反方向設定關聯。

▼ `@JoinColumn` 預設在多方 (**Many-Side**)：當你使用 **單向 `@ManyToOne`** 關聯時，`@JoinColumn` 會自動在 **多方 (子表, Many Side)** 來存放外鍵 (FK)。

```
@Entity
public class Child {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "parent_id") // 外鍵欄位
    private Parent parent;
}
```

▼ `mappedBy` 設定雙向關聯 (**One-To-Many & Many-To-One**)：`mappedBy` 必須定義在「**不擁有外鍵**」的一方 (通常是 **One-Side**)。`mappedBy` 的值對應到對方 (**Child**) 的 `@ManyToOne` 欄位名稱。

```
@Entity
public class Parent {
    @Id @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "parent") // mappedBy 指向 Child 類別的 parent 屬性
    private List<Child> children;
}

@Entity
public class Child {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "parent_id") // 外鍵欄位
    private Parent parent;
}
```

```

}

CREATE TABLE Parent (
    id BIGINT AUTO_INCREMENT PRIMARY KEY
);

CREATE TABLE Child (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    parent_id BIGINT, -- 外鍵
    FOREIGN KEY (parent_id) REFERENCES Parent(id)
);

```

▼ 單向 `@OneToMany` (外鍵仍然在多方, 但 `@JoinColumn` 設定在一方): 通常單向 `@OneToMany` 需要額外指定 `@JoinColumn`, 但外鍵仍會存放在多方 (Child)。

```

@Entity
public class Parent {
    @Id @GeneratedValue
    private Long id;

    @OneToMany
    @JoinColumn(name = "parent_id") // 外鍵仍然存在 Child 表
    private List<Child> children;
}

@Entity
public class Child {
    @Id @GeneratedValue
    private Long id;
}

CREATE TABLE Parent (
    id BIGINT AUTO_INCREMENT PRIMARY KEY
);

CREATE TABLE Child (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    parent_id BIGINT, -- 外鍵
    FOREIGN KEY (parent_id) REFERENCES Parent(id)
);

```

關聯類型	<code>@JoinColumn</code> 設定在哪	外鍵 (FK) 存在哪	<code>mappedBy</code> 設在哪
單向 <code>@ManyToOne</code>	Child.parent	Child (多方)	無
雙向 <code>@OneToMany</code> - <code>@ManyToOne</code>	Child.parent	Child (多方)	Parent.children
單向 <code>@OneToMany</code>	Parent.children	Child (多方)	無

📌 021 📌 `CascadeType.REMOVE` (連帶刪除)

▼ 如果 `Parent` 設定 `cascade = CascadeType.REMOVE`, 則刪除 `Parent` 會自動刪除關聯的 `Child`。

```

@OneToMany(mappedBy = "parent", cascade = CascadeType.REMOVE)
private List<Child> children;

```

```
session.delete(parent);

DELETE FROM Child WHERE parent_id = 1; -- 先刪除子表
DELETE FROM Parent WHERE id = 1; -- 再刪除父表
```

▼ 若沒有設定 `CascadeType.REMOVE`，則刪除 `Parent` 時，若 `Child` 仍存在外鍵指向 `Parent`，會發生 外鍵約束錯誤。

```
session.delete(parent);

DELETE FROM Parent WHERE id = 1;

org.hibernate.exception.ConstraintViolationException: Cannot delete or update a parent row: a foreign key constraint f
//=====
✅ 方法 1：先解除關聯，再刪除 Parent
for (Child child : parent.getChildren()) {
    child.setParent(null); // 解除關聯
}
session.delete(parent);
UPDATE Child SET parent_id = NULL WHERE parent_id = 1; -- 先解除關聯
DELETE FROM Parent WHERE id = 1; -- 再刪除父表
//=====
✅ 方法 2：設置 orphanRemoval = true
@OneToMany(mappedBy = "parent", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Child> children;
parent.getChildren().clear();
session.delete(parent);
```

▼ `orphanRemoval = true`（孤兒刪除機制）：在 `@OneToOne` 或 `@OneToMany` 關聯 中，當子物件不再被父物件參考時，它會被 自動刪除，不需要額外呼叫 `session.remove()` 或 `delete` 方法。

```
@Entity
public class Parent {
    @Id @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "parent", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Child> children = new ArrayList<>();

    public void removeChild(Child child) {
        children.remove(child); // 從 List 中移除，會自動 DELETE
        child.setParent(null);
    }
}

@Entity
public class Child {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "parent_id")
    private Parent parent;
}
```


機制	作用範圍	觸發條件	SQL 操作
<code>CascadeType.REMOVE</code>	刪除父物件時，自動刪除子物件	刪除父物件時	<code>DELETE FROM child WHERE parent_id = ?</code>
<code>orphanRemoval = true</code>	移除父物件關聯時，自動刪除子物件	子物件從集合中移除 或 <code>@OneToOne</code> 設為 <code>null</code>	<code>DELETE FROM child WHERE id = ?</code>

分層

📌 022 📌 資料存取物件 (Data Access Object,DAO)

DAO 是專門負責存取資料庫的類別，主要執行 **CRUD** 操作，但不負責商業邏輯。

- **分離業務邏輯與資料存取**：業務邏輯應該放在 **Service** 層，資料存取邏輯則放在 **DAO** 層。
- **提高可維護性**：如果需要更換資料庫技術（如 Hibernate, JPA, JDBC），只要修改 DAO，不影響其他層。
- **減少重複程式碼**：將常見的 **SQL 查詢與交易管理** 集中處理，方便重複使用。
- **符合 SOLID 原則**：單一職責 (**SRP**)：DAO 只負責資料存取，不處理業務邏輯。**開放封閉 (OCP)**：可以透過不同的實作來適應不同資料庫技術（如 Hibernate, JDBC）。

▼ 指 DAO 錯誤設計規範 (❌ DAO 內部管理交易)

```
public interface MemberDAO {
    void save(Member member); // 新增會員
    void update(Member member); // 更新會員
    void delete(int id); // 刪除會員
    Member findById(int id); // 透過 ID 查詢會員
    List<Member> findAll(); // 查詢所有會員
}

//=====
public class MemberDAOImpl implements MemberDAO {
    private SessionFactory sessionFactory;

    public MemberDAOImpl(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Override
    public void save(Member member) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();
        try {
            session.save(member);
            tx.commit();
        } catch (Exception e) {
            tx.rollback();
            throw e; // 轉拋例外
        } finally {
            session.close();
        }
    }
}
```

設計原則	說明
設計為介面 (Interface)	讓不同技術（如 JDBC、JPA、Hibernate）可以有不同實作
方法應為 <code>public</code>	讓其他程式可以存取 DAO 方法
方法應為非靜態方法	讓 DAO 可以管理 <code>SessionFactory</code> 或 <code>EntityManager</code>
發生 Checked Exception 時，應 <code>throws</code> 例外	避免業務層誤以為 DAO 方法成功執行
不應該顯示錯誤訊息	只拋出例外，不應該在 DAO 內 <code>System.out.println()</code>

❌ 錯誤做法 (DAO 管理交易)	✅ 正確做法 (Service 管理交易)
DAO 內部開啟 <code>beginTransaction()</code>	Service 層開啟 <code>beginTransaction()</code>
DAO 內部 <code>commit()</code> 交易	Service 層 <code>commit()</code> 交易
多個 DAO 方法無法組成同一個交易	多個 DAO 方法可以共用同一個交易
重複的交易管理程式碼	交易邏輯集中在 Service 層

```

    }

    @Override
    public Member findById(int id) {
        try (Session session = sessionFactory.openSession()) {
            return session.get(Member.class, id);
        }
    }
}
//=====
public class MemberService {
    private MemberDAO memberDAO;

    public MemberService(MemberDAO memberDAO) {
        this.memberDAO = memberDAO;
    }

    public void registerMember(String name, String email) {
        Member member = new Member(name, email);
        memberDAO.save(member);
    }

    public Member getMember(int id) {
        return memberDAO.findById(id);
    }
}

```

▼ **指DAO 正確設計規範**（交易應該由 Service 層管理）

```

public class MemberDAOImpl implements MemberDAO {
    private SessionFactory sessionFactory;

    public MemberDAOImpl(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Override
    public void save(Member member) {
        Session session = sessionFactory.getCurrentSession();
        session.save(member);
    }

    @Override
    public Member findById(int id) {
        Session session = sessionFactory.getCurrentSession();
        return session.get(Member.class, id);
    }
}
//=====
public class MemberService {
    private MemberDAO memberDAO;
    private SessionFactory sessionFactory;
}

```

```

public MemberService(MemberDAO memberDAO, SessionFactory sessionFactory) {
    this.memberDAO = memberDAO;
    this.sessionFactory = sessionFactory;
}

public void registerMember(Member member, Account account) {
    Transaction tx = sessionFactory.getCurrentSession().beginTransaction();
    try {
        memberDAO.save(member);
        accountDAO.save(account); // 這裡可能存取不同
        tx.commit(); // ✅ 只有當所有操作成功後才提交
    } catch (Exception e) {
        tx.rollback(); // ❌ 若發生錯誤，則回滾整個交易
        throw e;
    }
}
}

```

- **確保跨 DAO 方法的 ACID 特性**：如果在 DAO 內開啟/關閉交易，則無法在 Service 層統一管理跨 DAO 方法的交易，可能導致資料不一致。
- **避免重複的交易管理程式碼**：如果 DAO 內部開啟交易，每個 DAO 方法都會重複寫相同的交易控制邏輯，這樣會降低可維護性。
- **DAO 只負責資料存取，不負責交易管理**：DAO 主要負責單一表格的 CRUD 操作，而交易可能涉及多個 DAO 方法，因此應該在 **Service 層** 控制。

📌 023 📌 Service 類別 (Service Class)

Service 類別負責**業務邏輯**，呼叫多個 DAO 方法來完成一個完整的操作。因為一個 Service 方法通常包含多個資料庫操作，需要確保它們在**同一個交易內執行**，確保 **ACID 原則**。

- **負責業務邏輯** (Business Logic)。
- **呼叫多個 DAO 方法來完成一個完整操作** (例如客戶訂購商品需要寫入訂單表、扣除庫存表等)。
- ▼ **管理交易 (Transaction)**，確保所有 DAO 方法在**同一個交易內執行**，要嘛全部成功 (`commit()`)，要嘛全部失敗 (`rollback()`)。


```


@Service
@Transactional
public class OrderService {
    private final OrderDAO orderDAO;
    private final InventoryDAO inventoryDAO;

    @Autowired
    public OrderService(OrderDAO orderDAO, InventoryDAO inventoryDAO) {
        this.orderDAO = orderDAO;
        this.inventoryDAO = inventoryDAO;
    }

    public void placeOrder(Order order, Inventory inventory) {
        orderDAO.save(order);
        inventoryDAO.updateStock(inventory);
    }
}

```

```
//  `@Transactional` 會自動管理交易
}
}
```

▼  需求符合交易 (Transaction) 的概念，因為所有的操作 (檢查未付款金額、檢查庫存、更新庫存、寫入訂單) 都需要確保 **全部成功才提交**，否則就回滾 (rollback) 所有變更。

OrderServiceImpl.persistOrder 的設計

- `MemberService.checkUnpaidAmount()`：檢查客戶未付款金額是否超過限額
- `OrderService.checkStock()`：檢查庫存是否足夠
 - 若庫存不足，則丟出例外，**交易回滾**
 - 若庫存足夠，則更新庫存
- `OrderDAO.persistOrder()`：將訂單 (包含 `order` 和 `order_items`) 存入資料庫
 - 若發生例外則回滾交易
- 所有步驟都成功，才 `commit()` 交易

```
@Service
@Transactional
public class OrderServiceImpl implements OrderService {
    private final OrderDAO orderDAO;
    private final OrderItemDAO orderItemDAO;
    private final MemberService memberService;

    @Autowired
    public OrderServiceImpl(OrderDAO orderDAO, OrderItemDAO orderItemDAO, MemberService memberService) {
        this.orderDAO = orderDAO;
        this.orderItemDAO = orderItemDAO;
        this.memberService = memberService;
    }

    @Override
    public void persistOrder(Order order) {
        // 1 檢查客戶未付款金額
        memberService.checkUnpaidAmount(order.getCustomer());

        // 2 檢查並更新庫存
        for (OrderItem item : order.getItems()) {
            orderItemDAO.checkStock(item);
            orderItemDAO.updateProductStock(item);
        }

        // 3 儲存訂單
        orderDAO.persistOrder(order);
        //  `@Transactional` 會自動管理 `commit()` 或 `rollback()`
    }
}
```

其他

📌 024 📌 延遲載入 (Lazy Loading) & 立即載入 (Eager Loading)

延遲載入 (Lazy Loading) 與 立即載入 (Eager Loading) 是關於關聯資料何時從資料庫中擷取的策略。

▼ **FetchType.LAZY** (延遲載入)：當使用 **FetchType.LAZY** 時，**Hibernate** 不會立即載入關聯物件，而是等到第一次訪問該屬性時才從資料庫讀取資料。✅ 適用於：**@OneToMany**、**@ManyToMany**

```
@Entity
public class Parent {
    @Id @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "parent", fetch = FetchType.LAZY) // 延遲載入
    private List<Child> children;
}

@Entity
public class Child {
    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "parent_id")
    private Parent parent;
}

// Problem
Session session = sessionFactory.openSession();
Parent parent = session.get(Parent.class, 1L); // 只載入 Parent，不載入 Child
session.close();

List<Child> children = parent.getChildren(); // 這裡會拋出 LazyInitializationException
//resolved
List<Parent> parents = session.createQuery("SELECT p FROM Parent p JOIN FETCH p.children", Parent.class).getRes
```

💡 優點：

- 減少不必要的查詢，若程式只需要「單方」的屬性，可以避免讀取關聯物件，節省查詢時間與記憶體。
- 適合大規模資料關聯，例如一個 **Order** 可能關聯大量 **OrderItem**，使用 **Lazy** 可以避免一次載入所有訂單項目。

⚠️ 缺點：

- 當 **Session** 關閉後，嘗試存取 **Lazy** 屬性會拋出 **LazyInitializationException**，這在 **Web 應用程式 (Spring Boot, JPA)** 中是常見問題。
- 可能會導致 **N+1** 查詢問題，即：查詢單方物件時發出 1 次查詢，存取多方屬性時，為每個物件發出額外 **N** 次查詢，這可能導致效能下降

▼ **FetchType.EAGER** (立即載入)：當使用 **FetchType.EAGER**，**Hibernate** 會在查詢時立即載入關聯物件，不需要額外發出 SQL 查詢。

✅ 適用於：**@OneToOne** (一對一) **@ManyToOne** (多對一)

```
@Entity
public class Employee {
    @Id @GeneratedValue
    private Long id;

    @OneToOne(fetch = FetchType.EAGER) // 立即載入
    private Address address;
}

@Entity
```

```
public class Address {
    @Id @GeneratedValue
    private Long id;

    private String city;
}

Employee emp = session.get(Employee.class, 1L); // 同時載入 Employee 和 Address
```

💡 優點：

- 避免 `LazyInitializationException`
- 查詢時一次載入所有關聯資料，減少 N+1 查詢問題

⚠️ 缺點：

- 若不需要關聯資料，仍然會載入，可能造成不必要的效能開銷
- 使用 EAGER 可能導致 Hibernate 在 JOIN 時載入過多資料，影響效能

需求	適合的 <code>FetchType</code>	原因
需要一次載入關聯資料，避免 <code>LazyInitializationException</code>	<code>FetchType.EAGER</code>	立即載入可避免 session 關閉問題
可能不會用到關聯資料，想節省效能	<code>FetchType.LAZY</code>	減少不必要的 SQL 查詢，提高效率
<code>@OneToOne</code> / <code>@ManyToOne</code> (少量資料)	<code>FetchType.EAGER</code>	這類關聯通常不會有大量資料，一次載入較適合
<code>@OneToMany</code> / <code>@ManyToMany</code> (大量資料)	<code>FetchType.LAZY</code>	避免載入過多資料，影響效能
避免 N+1 查詢問題	<code>FetchType.LAZY</code> 搭配 <code>JOIN FETCH</code>	<code>JOIN FETCH</code> 讓 Hibernate 只用 1 次 SQL 取回所有資料

<code>FetchType</code>	適用關聯	優點	缺點
Lazy (延遲載入)	<code>@OneToMany</code> , <code>@ManyToMany</code>	減少不必要的查詢，提高效率	可能! <code>Lazy</code>
Eager (立即載入)	<code>@OneToOne</code> , <code>@ManyToOne</code>	查詢時一次載入，避免 <code>LazyInitializationException</code>	可能! 影響?

📌 025 📌 HQL 語法 (Hibernate Query Language, HQL)

HQL 是 Hibernate 框架提供的一種面向對象的查詢語言，類似 SQL，但它針對的是 Java 的 Entity (實體類)，而不是直接操作資料庫表格。

👉 HQL 主要用於查詢 Hibernate 管理的實體對象，而不是資料庫中的行列數據。

特點	HQL	SQL
操作對象	Entity 類別	資料庫表格
查詢語法	<code>FROM Member</code>	<code>SELECT * FROM member</code>
屬性名稱	Java 類別屬性 (<code>m.name</code>)	資料庫欄位 (<code>m.NAME</code>)
物件導向	✅ 支援	❌ 不支援
內建關聯	✅ 支援 (<code>JOIN FETCH</code>)	❌ 需要手動指定 <code>JOIN</code>
可移植性	✅ 跨 DBMS	❌ 依賴特定資料庫

```
String hql = "FROM Member m WHERE m.status = :status";
Query<Member> query = session.createQuery(hql, Member.class);
query.setParameter("status", "ACTIVE");
List<Member> members = query.getResultList();
```

✅ 面向物件 (使用 Entity 而非資料表) ✅ 與資料庫無關 (可適用不同 DBMS) ✅ 內建關聯 (支援 JOIN FETCH)

✅ 支援快取 (整合 Hibernate Cache) ✅ 簡化查詢 (比 SQL 更易讀)

- 如果你在使用 Hibernate，HQL 絕對是比 SQL 更好的選擇！

📌 026 📌 Logback 日誌框架 (Logback)

Logback 是 Java 生態中常用的日誌框架之一，它是 Log4j 的改進版，並且是 SLF4J (Simple Logging Facade for Java) 的預設實作之一。

1. **性能優異**：相較於 Log4j，Logback 擁有更快的日誌寫入速度，並且佔用更少的記憶體資源。
2. **自動重新載入 (Reconfiguration on the fly)**：當日誌設定檔變更時，Logback 能夠自動重新載入配置，無需重啟應用程式。
3. **更靈活的日誌管理**：支援 **基於時間或大小的日誌輪轉 (Rolling Policy)**，可以自動壓縮舊的日誌文件，避免佔用過多磁碟空間。
4. **內建 XML 和 Groovy 配置支持**：可以使用 `logback.xml` 或 `logback.groovy` 來配置日誌。
5. **與 SLF4J 無縫整合**：開發者可以透過 SLF4J API 使用 Logback，而不直接依賴 Logback 的 API，讓應用程式更具可擴展性。
6. **內建異常追蹤功能**：可以自動偵測應用程式內的異常，並記錄完整的堆疊資訊 (Stack Trace)。

▼ **指**範例與設定

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class LogbackExample {
    private static final Logger logger = LoggerFactory.getLogger(LogbackExample.class);

    public static void main(String[] args) {
        logger.info("這是一條 INFO 日誌");
        logger.warn("這是一條 WARN 日誌");
        logger.error("這是一條 ERROR 日誌", new RuntimeException("測試異常"));
    }
}
```

```
<dependencies>
<!-- SLF4J API (用於統一日誌框架的接口) -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>2.0.7</version>
</dependency>

<!-- Logback 核心 (實作 SLF4J) -->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.4.11</version>
</dependency>
</dependencies>

//=====
<configuration>
<!-- 定義日誌輸出格式 -->
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
        <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
</appender>

<!-- 設定根日誌記錄器 -->
<root level="info">
    <appender-ref ref="STDOUT"/>
</root>
</configuration>
```

```
</root>  
</configuration>
```
