Spring Boot 效能優化

載入策略與 N+1 問題指南

◆ 專業後端開發教學 ◆

◆ 基礎問題測試

當執行 `studentDAO.findById(11346099)` 時，
關聯的「選修課程」何時會被載入？

1. 查詢學生時一併載入
2. 存取課程屬性時才載入
3. 視 Hibernate 配置而定

錯誤的選擇將直接影響系統效能。

◆ 什麼是 ORM？(核心概念)

Object-Relational Mapping 是一座橋樑：

◆ 物件端 (Java)

- Class / Instance
- Field / Method
- 邏輯模型

◆ 關係端 (DB)

- Table / Row
- Column / Type
- 資料模型

> ***本質***：*透過「映射」讓開發者能用處理物件的方式操作資料庫。*
> ***代價***：*這種抽象會掩蓋真實的 SQL 行為，導致效能陷阱。*

◆ 純SQL

```php
<?php
$db = new PDO('mysql:host=127.0.0.1;dbname=test;charset=utf8', 'root', '');
$sql = $db->prepare('SELECT * FROM `users` WHERE `account`=:act');
$sql->bindValue('act', $_POST['account']);
$sql->execute();
$user = $sql->fetch(PDO::FETCH_ASSOC);
if (!$user) {
    header('location: login.php?err=找不到帳號');
}
```

◆ ORM

```
1  Product product = productRepository.findById(id)
2      .orElseThrow(() -> new NotFoundException("産品不存在"));
3
4  product.decreaseStock(quantity);
5
6  productRepository.save(product);
```

◆ 核心課題：N+1 問題

低效率的迴圈範例：

```java
List<Student> list = studentRepo.findAll(); // 1 次查詢
for (Student s : list) {
    s.getCourse().getName(); // 觸發 N 次查詢
}
```

後端日誌呈現：

產生了 1 + N 條 SQL 語句。

這是導致資料庫負載過高的主因。

Eager vs Lazy 載入邏輯

Eager Loading

- **立即載入關聯資料**
- 主體與關聯資料一次到位
- 適用於關聯資料極少
- 缺點：易造成記憶體浪費

Lazy Loading

- **延遲載入關聯資料**
- 使用 Proxy 代理物件
- 僅在真正呼叫時執行查詢
- 專業開發者的首選配置

◆ JPA 預設行為警示

請務必留意 JPA 的預設配置陷阱：

- **Eager (預設)**：`@ManyToOne`, `@OneToOne`
- **Lazy (預設)**：`@OneToMany`, `@ManyToMany`

實務建議：

明確將所有關聯宣告為 `FetchType.LAZY`。

◆ Lazy Loading 的風險管理

異常：LazyInitializationException

```java
// Service 層執行查詢
Student s = studentRepo.findById(1);

// Controller 層存取關聯
return s.getCourse().getName(); // 失敗：Session 已關閉
```

原因分析： 資料庫 Session 已經關閉。

標準解法： ~~使用 @Transactional 標註~~ 包事務。

◆ Eager 配置的誤區

注意： 改成 Eager Loading 並不能解決 N+1 問題。

當執行 `findAll()` 時，
JPA 會先取得主體清單，再個別補齊關聯資料。
結果同樣會噴出大量查詢語句。

◆ 最佳實踐： `JOIN FETCH`

在 Repository 層級優化查詢邏輯：

```java
@Query("SELECT s FROM Student s JOIN FETCH s.course")
List<Student> findAllWithCourse();
```

優點：

1. 透過一次 SQL Join 操作取得完整資料。
2. 大幅降低與資料庫的互動次數
3. JPQL (Java Persistence Query Language)

◆ 進階配置： `@EntityGraph`

若偏好宣告式管理，可採用此方案：

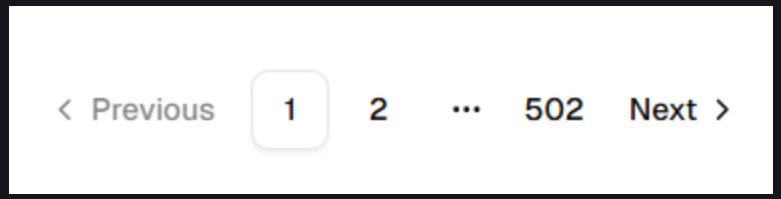
```
@EntityGraph(attributePaths = {"course"})
List<Student> findAll();
```

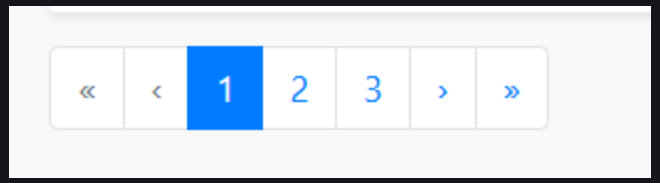
- 保持程式碼簡潔易讀。
- 靈活控制不同場景的載入深度。
- 建議搭配效能監控工具進行驗證。
- 就算沒寫 SQL，也會自動執行 Left Join

◆ 什麼是分頁查詢 (Pagination)？

核心概念：按需取貨，避免記憶體爆炸

- **定義**：當資料庫有萬筆資料時，不一次全部抓取，而是透過 `LIMIT` 與 `OFFSET` 語法，每次只抓取一小部分（如每頁 20 筆）。
- **物理意義**：減少資料庫 I/O 負擔，降低網路傳輸量，並保護應用程式（JVM）不會發生 **OOM (Out of Memory)**。
- **JPA 實作**：使用 `Pageable` 與 `Page<T>` 介面自動生成分頁 SQL。





◆ 前端如何發起分頁請求？

- **主要參數**：
  - `page`：第幾頁（注意：後端 JPA 預設從 0 開始）。
  - `size`：每頁幾筆。
  - `sort`：排序欄位與方向（如 `id,desc`）。

```
// 請求第 1 頁，每頁 10 筆，按建立時間倒序
axios.get('/api/orders', {
  params: { page: 0, size: 10, sort: 'createdAt,desc' }
});
```

```
https://localhost/api/orders?page=0&size=10&sort=createdAt,desc
```

◆ 後端 Controller 如何接收？

- **自動注入**：Spring Boot 會自動將 URL 中的參數封裝進 `Pageable` 物件。

```java
@GetMapping("/api/orders")
public Page<OrderDTO> getOrders(Pageable pageable) {
    // 直接將 pageable 傳給 Service
    return orderService.findAll(pageable);
}
```

- **預設值**：可以使用 `@PageableDefault(size = 20, sort = "id")` 指定行為。

◆ 回傳給前端的資料結構

核心物件：`Page<T>` 的封裝

```
{
  "content": [ ... ],        // 這一頁的實際資料
  "totalPages": 5,           // 總共有幾頁
  "totalElements": 48,       // 總共有幾筆資料
  "size": 10,                // 每頁幾筆
  "number": 0                // 當前頁碼
}
```

◆ 分頁遇到了 Fetch Join 會怎樣？

核心問題：記憶體分頁的陷阱

- **現象**：當你在 JPQL 使用 `JOIN FETCH` 抓取 `OneToMany` 關聯並分頁時，Hibernate 會丟出警告：`HHH000104: firstResult/maxResults specified with collection fetch; applying in memory!`。
- **後果**：Hibernate 會先從資料庫抓取 **所有** 資料，然後才在 Java 程式內手動過濾前 20 筆。
- **危險性**：這讓原本的分頁失去了意義，效能與沒分頁一樣慘，甚至更危險。

◆ 手動過濾就算了還會弄壞分頁

底層原因：一對多的「資料膨脹」

- **SQL 視角：** 一個 User 有 3 筆 Order，`JOIN` 後，結果會有 3 列記錄，同一個 User 的欄位重複出現。
- **計數困境：** 資料庫層級的 `LIMIT 5` 抓到的是「5 列 SQL 結果」，而不是「5 個 User」。
- **邏輯衝突：** JPA 無法在 SQL 層面判斷這 20 列到底代表幾個完整的實體，因此被迫將判斷邏輯移回 Java 記憶體處理。

SQL JOIN 的資料膨脹範例

```sql
SELECT u.id, u.name, o.id, o.amount
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
LIMIT 5;
```

> ***對應 JPQL：*** *`SELECT u FROM User u LEFT JOIN u.orders o`*
>
> *（Hibernate 會轉成上面的 SQL），但仍會遭遇同樣的分頁問題*

查詢結果：

USER ID	USER NAME	ORDER ID	ORDER AMOUNT
1	Alice	101	$100
1	Alice	102	$200
1	Alice	103	$150
2	Bob	201	$300
2	Bob	202	$250

> ***注意：****只有 2 個 User，但 SQL 回傳了 5 列記錄。若使用* `LIMIT 3` *，只會抓到 Alice 的前 3 筆訂單，Bob 完全被遺漏！*

◆ 共同的坑：笛卡兒積 (Cartesian Product)

無論用哪一種，都要注意以下兩點：

同時抓取多個 List 關聯： 這會導致資料庫回傳的資料量呈倍數成長（笛卡兒積），甚至丟出 `MultipleBagFetchException`。

分頁問題： 在對關聯物件進行 Fetch Join 時，若同時使用 `Pageable` 進行分頁，Hibernate 會將所有資料抓到記憶體中再進行分頁，這在資料量大時會導致 OOM (記憶體溢位)。

◆ 解法 A — batch_fetch_size

◆ 核心技巧：化整為零，分批抓取

- **原理**： 將原本的 N+1 轉化為 IN 查詢。
- **優勢**： 寫法最簡單（只需改配置），且完美支援 Spring Data JPA 的 Pageable。
- **展示**： 秀出兩段式的 SQL Log，證明它既解決了 N+1，又沒有弄壞分頁。

◆ 解法 A 實作：`batch_fetch_size` 與 `Page`

- **場景：** 查詢訂單（Order）時，需要顯示關聯的商品（Items）。

```
# application.yml
spring.jpa.properties.hibernate.default_batch_fetch_size: 100
```

```
// 不要寫 JOIN FETCH，讓 Batch Size 處理
Page<Order> findAll(Pageable pageable);
```

結果：

1. 第一條 SQL：`SELECT ... FROM orders LIMIT 10`
2. 第二條 SQL：`SELECT ... FROM items WHERE order_id IN (?, ?, ...)`

◆ 解法 B — 兩段式 DTO 查詢

核心技巧：精確控制，徹底分離

- **步驟：**
  i. 先用分頁查詢抓出主體的 **ID 清單**。
  ii. 再拿這組 ID 去執行一個帶有 `JOIN FETCH` 的查詢，抓出完整的實體圖。
- **優點：** 繞過了 Hibernate 的記憶體分頁警告，且生成的 SQL 非常乾淨。
- **適用場景：** 高併發系統，或對 SQL 效能有極致要求的場景。

◆ 解法 B 實作

流程只有三步

```
// 1) 先分頁查 ID（保留 DB 分頁）
Page<Long> idPage = dao.findPageIds(pageable);

// 2) 再用 ID 批次查完整資料（一次抓關聯，避免 N+1）
List<Order> orders = dao.findAllWithItemsByIdIn(idPage.getContent());

// 3) 轉 DTO 並回傳 Page
return new PageImpl<>(toDto(orders), pageable, idPage.getTotalElements());
```

什麼時候用？

- 你需要「分頁準確」又要「避免 N+1」。
- 資料量大，**高併發**。

◆ 效能優化總結

1. **落實全 Lazy 策略**：避免系統自動過度負載。
2. **具備 N+1 敏感度**：在開發階段預判查詢壓力。
3. **善用分析工具**：開啟 SQL Log 觀察真實行為。
4. **持續驗證與重構**：確保程式碼的執行效率。

技術深度體現於對細節的處理與堅持。

Q & A

感謝聆聽

◆ 結束 ◆

◆ 課後測驗（是非題）

請判斷下列敘述是否正確：

1. `@OneToMany` 在 JPA 預設是 Lazy 載入。
2. 只要把關聯改成 Eager，就能完全避免 N+1 問題。
3. `LazyInitializationException` 常見原因是 Session 關閉後才存取延遲載入欄位。

◆ 課後測驗（單選題）

1) 當你在 `OneToMany` 關聯上使用 `JOIN FETCH` 並搭配 `Pageable`，最可能出現什麼情況？

A. Hibernate 自動把 SQL 最佳化成最少查詢，且分頁一定正確。

B. 可能出現 `HHH000104`，並在記憶體做分頁。

C. 會直接丟出 `ClassNotFoundException`。

D. 只要加上 `@Transactional` 就不會有任何效能問題。

◆ 課後測驗（多選題）

1) 下列哪些做法有助於避免 N+1，同時兼顧可擴展性？（可複選）

A. 依場景使用 `JOIN FETCH` 或 `@EntityGraph`。

B. 在大量分頁場景優先考慮 `batch_fetch_size`。

C. 所有關聯一律改成 Eager，減少代理物件。

D. 使用兩段式查詢（先查 ID，再批次查完整資料）。

◆ 講者參考答案（可上課後再揭示）

- 是非題：1. 對 2. 錯 3. 對
- 單選題：1) B
- 多選題：1) A、B、D