
Nguyen Hong Son @nhs3108 Follow
★ 910 50 22

Published Jan 9th, 2018 9:18 PM - 9 min read

10 sai lầm phổ biến làm giảm hiệu suất ứng dung khi sử dụng Hibernate - Phần 1

Java Hibernate

Hello các bạn! Hôm nay có thể coi là hạn cuối viết report tháng của mình. Cả tháng rồi

VIBLO Posts Questions Discussions Search Viblo Sign In/Sign up

hiệu suất ứng dung khi sử dụng Hibernate". Bài viết được dịch và chỉnh sử từ 10 Common Hibernate Mistakes That Cripple Your Performance .Hi vọng sau bài viết này, các bạn có thể đúc rút ra 1 số kinh nghiệm cũng như lưu ý hơn khi sử dụng Hibernate - 1 ORM rất phổ biến trong lập trình các ứng dụng Java

Nếu như các bạn chưa đọc qua series bài viết Hibernate Caching toàn tập - Hướng dẫn có source code demo thì hãy ngó qua đôi chút nhé.

1. Sai lầm số 1: Sử dụng Eager Fetching.

FetchingType định nghĩa khi nào Hibernate khởi tạo 1 liên kết (`association`). Bạn có thể chỉ ra điều đó với thuộc tính fetch của @OneToMany, @ManyToOne, @ManyToMany và @OneToOne annotation.

```java
@Entity
public class Author{

    @ManyToMany(mappedBy="authors", fetch=FetchType.EAGER)
    private List<Book> books = new ArrayList<Book>();

    ...
```


+4

Đúng như cái tên của nó. (`Eager` : háo hức). Với `fetch = FetchType.EAGER` , Hibernate sẽ load các liên kết khi nó load một thực thể. Ví dụ, khi Hibernate load một thực thể Author, nó cũng kéo về thông tin của các thực thể Book có liên kết tới Author đó. Thử nghĩ xem, nếu bạn cần thực hiện truy vấn để lấy thông tin 1000 tác giả để phục vụ cho việc in danh thiếp cho họ, nhưng lệnh truy vấn đó của bạn ngoài việc lấy thông tin của các tác giả đó, lại đi lục lọi thông tin và hàng chục cuốn sách tương ứng của mỗi ông. Việc làm ấy chẳng phải rất vô nghĩa hay sao? Phương pháp này trong trường hợp vừa đề cập ở trên thực sự kém hiệu quả. Tốt hơn hết chúng ta nên thay thế bằng FetchType.LAZY. Nó sẽ trì hoãn (delay) việc khởi tạo quan hệ đó cho đến khi ta cần (`q1`). Điều đó tránh việc rất nhiều truy vấn không cần thiết, giúp hiệu năng ứng dụng của bạn được tăng lên.

FetchType mặc định JPA

```
OneToMany: LAZY
ManyToOne: EAGER
ManyToMany: LAZY
OneToOne: EAGER
```

Hibernate tôn trọng các giá trị mặc định đó nhưng khuyến nghị không sử dụng EARGE

VIBLO Posts Questions Discussions Search Viblo Sign In/Sign up

+4

> The Hibernate recommendation is to statically mark all associations lazy and to use dynamic fetching strategies for eagerness. This is unfortunately at odds with the JPA specification which defines that all one-to-one and many-to-one associations should be eagerly fetched by default. Hibernate, as a JPA provider, honors that default.

Vậy, câu hỏi đặt ra là, khi bạn sử dụng FetchType.LAZY, thì khi cần (mà mình đánh dấu `q1` ở trên) ta phải làm thế nào? Thắc mắc đó mình sẽ giải thích cho các bạn ở 1 bài viết

khác. Hoặc ngay bây giờ, bạn google là ra liền à 😄

2. Sai lầm số 2: Bỏ qua FetchType mặc định của liên kết x-To-One

Một điều tiếp theo mà bạn cần phải làm để ngăn chăn việc EARGE FETCHING là thay đổi FetchType mặc định cho tất cả liên kết x-To-One. Bao gồm @ManyToOne và @OneToOne bởi thật không may, như mình đã nói phía bên trên, fetch type mặc định của 2 kiểu liên kết đó đều là EARGE.

```
ManyToOne: EAGER
```


Trong một số trường hợp, việc mặc cho các kiểu liên kết x-To-One sử dụng EARGE FETCHING cũng không phải vấn đề quá lớn bởi vì ta sẽ chỉ load thêm 1 record trong database, Tuy nhiên nếu ta cần lấy thông tin để biết giá tiền 1 cuốn sách, thì ta không cần phải mất công sức lật tìm trang sách nào đó để biết tác giả là ai dù điều đó không làm mấy mất công. Nhưng nếu là 1000, 10.000 v.v.v thì nó cũng thành vấn đề đấy. Đúng ko nào?

Vì thế, để chắc chắn rằng ta sử dụng LAZY FETCHING cho @ManyToOne và @OneToOne , ta hãy luôn chỉ định fetch type cho nó nhé. Ví dụ

```
@Entity
public class Review {

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "fk_book")
    private Book book;
    ...
}
```

3. Sai lầm số 3: Không khởi tạo các liên kết cần

Khi bạn sử dụng FetchType.LAZY cho tất cả các mối liên kết để tránh sai lầm số 1 và số 2 mà mình đã đề cập ở trên, bạn sẽ tìm ra 1 vài vấn đề N+1 trong code của mình (N+1 query là gì, mình tin rằng dân sử dụng cơ sở đều biết và đều cần phải biết. Vì thế nếu bạn nào chưa biết, vui lòng google giúp mình nhé). Tuy nhiên, mình sẽ lấy 1 ví dụ đơn

giản về vấn đề N+1 như sau. Bằng cách nào đó, bạn lấy được 1 list n tác giả. Lúc này, dựa vào các tác giả bạn lấy được vừa rồi, bạn cần lấy thông tin về tất cả các cuốn sách của từng tác giả đó. khi đó query sẽ kiểu như thế này

```
List<Author> authors = em.createQuery("SELECT a FROM Author a", Author.class).getResultl
for (Author a : authors) {
    log.info(a.getFirstName() + " " + a.getLastName() + " wrote "
            + a.getBooks().size() + " books.");
}
```

Như vậy, với n tác giả, n vòng lặp, n lần truy vấn DB. Vì sao? Vì chúng ta để là lazy loading. Vì vậy nó sẽ không lấy thônng tin các sách tại thời điểm lấy được thông tin tác giả. Do đó nên trong mỗi vòng lặp kia lại thực hiện truy vấn tới db lấy thông tin về các sách. Mệt phết đúng ko? Cũng bình thường thôi =)) Bạn có thể tránh sai lầm trên 1 cách dễ ràng. đó là ra lệnh cho Hibernate khởi tạo các quan hệ cần thiết. Có nhiều cách để làm điều đó, (mình sẽ chỉ ra ở 1 bài viết riêng về FETCH TYPE) nhưng một trong những


```
Author a = em.createQuery(
                "SELECT a FROM Author a JOIN FETCH a.books WHERE a.id = 1",
                Author.class).getSingleResult();
```


4. Sai lầm số 4: SELECT nhiều record hơn bạn cần.

Cái này rất dễ hiểu, rất dễ để không lặp lại. Nhưng đúng là khi review code của các bạn trong team, mình thấy các bạn lại dễ mắc phải, nhất là các bạn mới ra trường và trải nghiệm dự án thực tế. Ví dụ, yêu cầu bài toán là lấy ra 5 author có id nhỏ nhất. Nhiều bạn làm như sau

```
List<Author> authors = em.createQuery("SELECT a FROM Author a ORDER BY a.id ASC", Autho
List<Author> authors result = authors = authors.stream().limit(5);
```

Như vậy, dữ liệu tại statement 1 lấy ra tất cả các n record author. Sau đó bạn dùng java để limit lấy 5. Thử hình dung với số lượng record author rất lớn, thì hiệu suất sẽ ra sao nhỉ? Rất tệ đúng k? Trong trường hợp này, điều cần làm chỉ là

```
                                .getResultList();
```

+4

5. Sai lầm số 5: Không sử dụng Bind Parameters

Mình đã từng làm 1 dự án dính một lỗi này dù anh bạn ấy ko cố ý mà chỉ vì lười thôi. Đó là khi init project, a có viết 1 function để lấy thông tin user dự vào username-password. Anh không sử dụng Bind Parameter mà lại đi cộng chuỗi như thế này

```
String query = "SELECT * FROM users u WHERE u.username = '" + inputUserName + "'  AND u
...
```

Ngay khi khách hàng vô tình phát hiện được đã claim cho 1 trận và nghi ngờ năng lực làm việc của a ta, cũng như của công ty bên mình. Vì sao à? Đơn giản thôi, câu lệnh trên sử dụng chính dữ liệu mà người dùng nhập vào, gây ra sơ hở nghiêm trọng liên quan tới SQL Injection.

Việc sử dụng Bind Parameter cung cấp nhiều lợi ích không liên quan tới performance

VIBLO Posts Questions Discussions Search Viblo Sign In/Sign up

- thực hiện chuyển đổi tự động (các kiểu dữ liệu).
- ngăn chặn lỗ hổng chèn mã SQL- SQL Injection

Tạm thế đã. Còn lại mình sẽ trình bày trong phần 2 nhé.

Related

Những lý do khiến ta chọn Hibernate thay vì JDBC
Nguyen Van Vu
9 min read
6133 5 1 4

Hibernate Caching - Bài 1: First Level Cache
Nguyen Hong Son
7 min read
2013 4 1 4

Một số mẹo viết câu truy vấn hiệu quả
Nguyen My Huyen
6 min read
248 3 0 5

Eager Loading trong laravel sử dụng with() hay load()?
Nguyễn Thanh Hải
2 min read
2517 2 0 3

More from Nguyen Hong Son

- JWT - Từ cơ bản đến chi tiết — Nguyen Hong Son
- 5 mẹo để tối ưu câu truy vấn SQL của bạn — Nguyen Hong Son
- Làm việc với Email trong môi trường development. — Nguyen Hong Son
- Sơ lược về VIEW và MATERIALIZED VIEW trong... — Nguyen Hong Son

Comments

VIBLO Posts Questions Discussions Search Viblo Sign In/Sign up

Vĩnh Thắng @vinhthang — Jan 10th, 2018 10:18 AM

ko liên quan nhưng trong sai lầm số 5 thì công ty đổ tội cho 1 anh, khi đã bàn giao gì đấy cho khách thì trách nhiệm là của cty, ko phải của anh ấy ạ

0 | Reply Share