休眠

JDBC 允许我们连接到数据库并提取数据，但是我们必须执行大量手动编码来检查和转换类型、处理结果集和异常等。我们可以将其中的大部分抽象为一个服务类，例如 `DataService` at最后一页的末尾，但我们仍然需要编写数据服务 - 尽管它是很多样板代码，我们几乎需要从每个新类的模板中复制粘贴。我们可以编写一个脚本，以某种机器可读的格式给出 ER 图，自动为此类生成数据服务来自动执行所有这些操作 - 或者我们可以使用 Hibernate 来为我们完成此操作。

Hibernate 是一个对象关系映射（ORM）框架，也就是说，它在运行时自动生成一种高级形式的数据服务，其中包括许多额外的功能，例如会话、事务、缓存和连接池。它实现了 Java Persistence Api (JPA)，这是一个基于 JDBC 构建的 API，用于实现 ORM。实际上Hibernate有它自己超越JPA的特性，比如它自己的查询语言。

在实际应用程序中，您大部分时间都会使用 ORM，但您可以使用 SQL 来实现 ORM 无法轻松支持的更高级查询。

示例应用程序 - 设置

`code/orm` 单元存储库中有一个示例应用程序。

POM 文件只是对 Hibernate 有一个额外的依赖：

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.27.Final</version>
</dependency>
```

Hibernate 的配置位于 `src/main/resources/hibernate.cfg.xml` . Hibernate 使用*会话工厂*来创建会话，您可以在其中进行查询：

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>

    <!-- These properties set up the database connection. -->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="connection.url">jdbc:mariadb://localhost/elections?
localSocket=/var/run/mysqld/mysqld.sock</property>
    <property name="connection.username">vagrant</property>

    <!-- Don't use this in production, use a connection pool instead. -->
    <property name="current_session_context_class">thread</property>

    <!-- Display generated SQL on the console. -->
    <property name="show_sql">true</property>

    <!-- The classes to map to database tables. -->
    <mapping class="org.example.Candidate" />
    <mapping class="org.example.Party" />
    <mapping class="org.example.Ward" />

  </session-factory>
</hibernate-configuration>
```

- 该*方言*选择 SQL 方言与数据库对话，在本例中为*MySQL*（没有单独的 MariaDB，因为两者在所有实际用途上都是相同的）。
- 连接字符串位于 中 `connection.url`，减去用户名/密码，因为它们有单独的属性。请注意，我们在这里包含了套接字选项。
- 用户名和密码（如果需要）位于单独的属性中。
- 连接池对于实际应用程序中的性能非常重要，但我们在这里不关心这一点，只说每个线程一个连接（我们在程序中不使用多个线程，因此它并不那么重要）。
- `show_sql` 是一个调试属性，它将所有生成的 SQL 打印到标准输出。在调试您自己的应用程序时了解这一点很有用。
- 最后，我们列出了希望 Hibernate 处理的所有类。在 SPE 中，您将使用 Spring 框架来自动处理此问题，但现在我们将它们全部列出。

这些类本身是标准的 Java 值类（私有字段、公共 getter/setter），用 JPA 注释进行修饰，以向 Hibernate 解释它们的工作原理：

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Party {
    @Id private int id;
    private String name;

    public Party() {}

    public int getId()      { return id; }
    public String getName() { return name; }

    public void setId(int id)          { this.id = id; }
    public void setName(String name)   { this.name = name; }
}
```

- `@Entity` 意味着这是映射到数据库中的表的东西。默认情况下，Hibernate 根据类名猜测表名，但您可以使用参数（例如 ）更改此设置 `@Entity(name="Parties")` 。
- `@Id` 表示主键。

候选类更有趣，因为它有外键：

```
@ManyToOne
@JoinColumn(name = "party")
private Party party;
```

- `@ManyToOne` 告诉 Hibernate 这是多对一关系的外键（还有 `@ManyToMany` ）。
- `@JoinColumn` 设置外键列的名称，此处默认为 `party_id` 。

应用示例-查询

让我们看一下主类（Example.java）。首先，Hibernate 使用会话工厂来管理自己的数据库连接和会话：

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

public class Example implements AutoCloseable {

  SessionFactory sessionFactory;

  public Example() {
    sessionFactory = new Configuration().configure().buildSessionFactory();
  }

  public void close() {
    sessionFactory.close();
  }

  // ...
}
```

您可以从全局 `Configuration` 类中获取一个，如果您愿意，您可以向该类传递参数以覆盖 XML 文件中的参数（例如，这可以让您更改设置以响应命令行参数）。会话工厂是一种资源，因此我们将自己的示例类也作为一种资源，这样我们就可以像这样运行它：

```
public static void main(String[] args) {
    try (Example example = new Example()) {
        example.run();
    }
  System.out.println("Done.");
  System.exit(0);
}
```

最后的 exit 命令可确保程序退出，即使 Hibernate 仍有后台线程运行。

在该 `run()` 方法中，我们使用 Hibernate 会话来管理数据库连接：

```
try (Session session = sessionFactory.openSession()) {
    // code
}
```

然后我们将提供三个使用 Hibernate 的示例。

通过ID加载实体

```
Party p1 = session.get(Party.class, 1);
System.out.println("    The party with id=1 is: " + p1.getName());
```

要通过 id 获取实例，我们只需 `get` 使用所需的类和 id 调用会话即可。传递类既告诉 Hibernate 从哪个表加载，又告诉编译器要返回什么类型的对象 - Java 泛型的工作方式，这让我们可以 `Party` 直接将结果分配给变量，而无需进行强制转换。

使用查询加载

```
TypedQuery<Ward> query = session.createQuery("FROM Ward", Ward.class);
List<Ward> wards = query.getResultList();
System.out.println("  Wards:");
for (Ward ward : wards) {
    System.out.println("    " + ward.getName());
}
```

一个 `TypedQuery` 对象接受 HQL 中的字符串、Hibernate 自己的查询语言（基于 SQL）和要返回的对象的类 - 这样 Java 编译器就可以计算出返回类型。 `getResultList` 以列表形式返回所有结果。

Advanced note

该 `TypedQuery` 接口是JPA的一部分，声明大致如下：

```
interface TypedQuery<T> {
  // ...
  List<T> getResultList();
}
```

泛型的使用允许编译器确保返回类型与您在创建查询时提供的类型参数相匹配。当然，您不会直接创建它，而是从 Hibernate 会话请求一个查询对象，但在幕后有一个 org.hibernate.query.internal.QueryImpl 以及许多其他 Hibernate 相关的实现类。

这些类可以将类型参数作为其构造函数的参数，但如果您查看源代码，它们不会：似乎 Hibernate 在内部进行了强制转换以获得正确的类型，只要 Hibernate 开发人员这样做就是安全的知道他们在做什么。

带连接和参数的查询

对于更复杂的示例，我们看以下内容：

```
TypedQuery<Candidate> q = session.createQuery("FROM Candidate c WHERE
c.party.name = :name", Candidate.class);
q.setParameter("name", "Labour");
List<Candidate> candidates = q.getResultList();
System.out.println("  Labour Candidates:");
for (Candidate c : candidates) {
    System.out.println("    " + c.getName() + " (" + c.getWard().getName() +
")");
}
```

HQL 与 SQL 一样，允许使用 WHERE 子句来过滤结果。它还允许准备好的语句，但比 JDBC 更好，因为参数可以有名称。您在查询字符串 ( ) 中使用冒号声明一个参数 `:name` ，然后使用 `setParameter(name, value)` 它来绑定它 - 编写此方法以便它可以采用任何类型的值，大概结合了 int/float 类型和 `Object` 其他所有类型的重载。

Hibernate 将自动执行必要的 JOIN 来获取与每个 Candidate 关联的一方，它为此查询运行的 SQL 在我的机器上如下所示：

```
select party0_.id as id1_1_0_, party0_.name as name2_1_0_ from Party party0_
where party0_.id=?

select candidate0_.id as id1_0_, candidate0_.name as name2_0_,
candidate0_.party as party4_0_, candidate0_.votes as votes3_0_,
candidate0_.ward as ward5_0_ from Candidate candidate0_
cross join Party party1_ where candidate0_.party=party1_.id and party1_.name=?
```

Hibernate 决定在这里执行两个查询（可能是并行的，因此语句在终端上打印的顺序可能不准确）。第一个是因为如果没有一方具有所提供的名称，那么 Hibernate 可以停止并返回一个空列表；如果存在，则继续第二个查询以连接两个表。

N+1问题

请注意，在上面的查询中，Hibernate 不会获取病房名称。这里并不重要，因为它们已经在前一个查询的 Hibernate 缓存中。但是，如果您注释掉前两个查询并只保留第三个查询（Example.java 中的第 41-50 行），则会发生可怕的事情：

```
select candidate0_.id as id1_0_, candidate0_.name as name2_0_,
candidate0_.party as party4_0_, candidate0_.votes as votes3_0_,
candidate0_.ward as ward5_0_ from Candidate candidate0_ cross join Party
party1_ where candidate0_.party=party1_.id and party1_.name=?
select party0_.id as id1_1_0_, party0_.name as name2_1_0_ from Party party0_
where party0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
```

Hibernate 正在为每个病房触发一个查询！这称为 N+1 问题，因为返回 N 个结果的一个 HQL 查询最终会生成 N+1 个 SQL 查询，这效率非常低，尤其是对于大型 N 而言。

查看循环结构：

```
TypedQuery<Candidate> q = session.createQuery("FROM Candidate c WHERE
c.party.name = :name", Candidate.class);
q.setParameter("name", "Labour");
List<Candidate> candidates = q.getResultList();
System.out.println("  Labour Candidates:");
for (Candidate c : candidates) {
    System.out.println("    " + c.getName() + " (" + c.getWard().getName() +
")");
}
```

从查询本身来看，Hibernate 不清楚是否需要病房名称，因此 Hibernate 默认情况下不会 JOIN 它们，如果您实际上不需要它们，这会更有效。然而，在底部的 for 循环中，您确实访问了名称，因此每次通过循环时，Hibernate 都被迫运行一个新查询来获取相关病房的名称。

Advanced note

Hibernate 如何触发 `.getWard().getName()` 您在 Candidate 和 Ward 类中实现的简单查询？

答案是，Hibernate 在运行时创建 Candidate 的代理子类并返回其实例，而不是实际的 Candidate 对象。当且仅当它们实际被调用时，这些代理对象已被 `getWard()` 重写以触发另一个查询。

这里的解决方案是在查询时告诉 Hibernate 您需要病房：

```
TypedQuery<Candidate> q = session.createQuery("FROM Candidate c JOIN FETCH
c.ward WHERE c.party.name = :name", Candidate.class);
```

这是“我要使用病房，所以也请执行 JOIN 来加载它们”的 HQL。Hibernate 现在再次对候选者使用单个查询：

```
select party0_.id as id1_1_0_, party0_.name as name2_1_0_ from Party party0_
where party0_.id=?

select candidate0_.id as id1_0_0_, ward1_.id as id1_2_1_, candidate0_.name as
name2_0_0_,
candidate0_.party as party4_0_0_, candidate0_.votes as votes3_0_0_,
candidate0_.ward as ward5_0_0_, ward1_.electorate as electora2_2_1_,
ward1_.name as name3_2_1_
from Candidate candidate0_
inner join Ward ward1_ on candidate0_.ward=ward1_.id
cross join Party party2_
where candidate0_.party=party2_.id and party2_.name=?
```

还有另一种方法可以解决这个问题：如果每次加载 Candidate 时，您都希望加载 ward 名称，那么您可以在 JPA 注释上声明：

```
@ManyToOne(fetch = FetchType.EAGER)
```

练习1

使用 Country、Region、County 和 Ward 表为人口普查数据库实现 JPA/Hibernate 示例应用程序（在本练习中忽略统计/职业）。例如，您可以在主程序中实现以下内容：

- 给定一个行政区代码，加载该行政区并打印出所有相关信息（行政区名称、县名等）。
- 给定一个行政区名称，打印出所有拥有该名称行政区的县。

注意避免第二种情况的N+1问题。

练习2

到目前为止，您所学到的知识将允许您在层次结构中向上导航，例如，给定一个行政区，您可以找到其相关的县。这个练习是相反的：给定一个县对象，您想要找到其中的所有行政区。

为此，请将以下属性添加到您的 County 类中：

```
@OneToMany(mappedBy = "county")
private List<Ward> wards;
```

的参数 `mappedBy` 必须是 Ward 类中包含县引用的字段的字段名称 - 您可能已 `parent` 在类中调用它或其他名称。

然后，为此列表属性添加 getter 和 setter。

您现在已经创建了所谓的*双向关联*：一个行政区包含一个县属性，一个县包含一个行政区列表。您可以在 Java 代码中双向导航。

编写一个查询，加载布里斯托尔市县 (E06000023)，并使用从县对象开始的 Java for 循环打印其所有行政区名称的列表。确保编写 HQL 查询，以免在此处造成 N+1 问题。

这个示例还有助于解释为什么我们默认情况下不急切获取所有内容：如果您使用双向关联来做到这一点，那么加载任何对象都会将整个数据库加载到内存中！