

Spring：领先一步 — Spring 数据访问（第 6 部分）

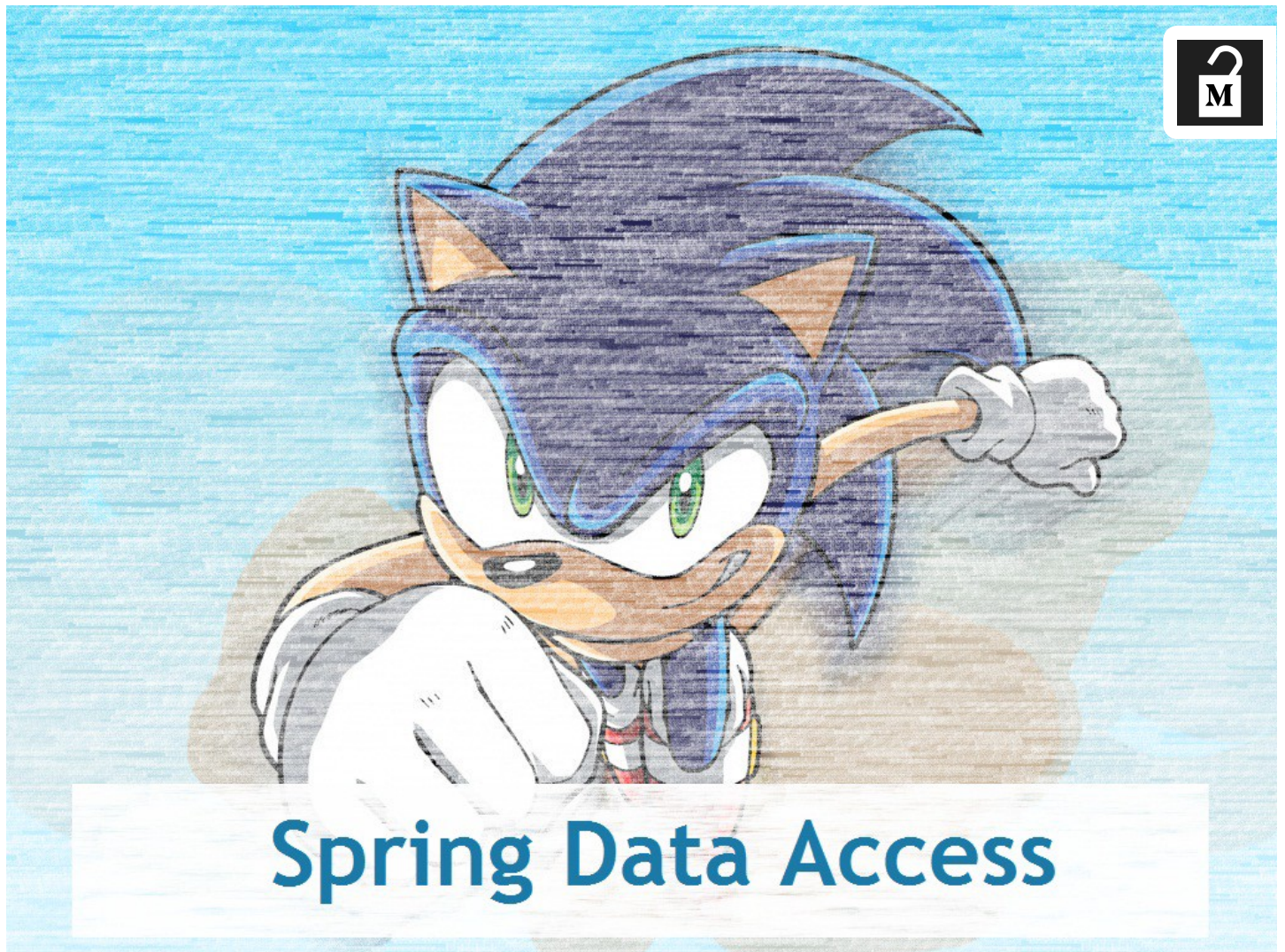
使用 JDBC 和 Hibernate 处理数据库。



奥马尔·埃尔加布里

跟随

2017 年 9 月 12 日 · 8分钟阅读



Spring：领先一步 — Spring 数据访问

设置环境

在开始之前，请确保您已具备以下条件：

1. 数据库服务器（XAMPP、WAMP、MySQL 等）
2. Hibernate JAR 文件（及其依赖项）和 JDBC 驱动程序。

如果您使用的是 maven，则在“pom.xml”文件中添加 hibernate 和 JDBC 驱动程序（在我们的示例中是 MySQL）依赖项。

```
<!-- 休眠 -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.1.7.Final</version>
</dependency>

<!-- - MySQL -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.40</version>
</dependency>
```



完成后，创建用户表，并生成一些假数据进行测试。

```
创建表`
users` (`id` int (11) NOT NULL AUTO_INCREMENT,
`name` varchar (60) NOT NULL,
`email` varchar (255) NOT NULL,
PRIMARY KEY (`id`)
)
```

JDBC

Spring JDBC 模块负责从打开连接、准备和执行 SQL 语句、处理异常、处理事务和最终关闭连接开始的所有底层细节。

我们需要做的只是定义连接参数并指定要执行的 SQL 语句。

但是，在使用 Spring 数据访问模块之前，让我们看看如何制作一个普通的旧 JDBC。

基本的 JDBC 连接

为了连接数据库并从数据库中获取数据，有 5 个基本步骤。

```
1 import java.sql.Connection;
2 import java.sql.DriverManager;
```

```
3  import java.sql.ResultSet;
4  import java.sql.Statement;
5
6  public class Test {
7      public static void main(String[] args) {
8          try {
9              // 1. Register JDBC Driver
10             Class.forName("com.mysql.jdbc.Driver");
11
12             // 2. Create Connection Object
13             Connection con = DriverManager.getConnection("jdbc:mysql://localhost/demoDB","root", "");
14
15             // 3. Create Statement object and Execute a Query
16             Statement stmt = con.createStatement();
17
18             // 4. Extract data from result set
19             ResultSet rs = stmt.executeQuery("select * from users");
20             while(rs.next()) {
21                 System.out.println(rs.getInt("id") + " " + rs.getString("name"));
22             }
23
24             // 5. Close the connection
25             con.close();
26
27             } catch (Exception e){
28                 // Handle errors here
29             }
30     }
31 }
```



Test.java hosted with ❤ by GitHub

[view raw](#)

因此，首先，您需要注册一个 JDBC 驱动程序，以便加载驱动程序的类文件并可以将其用作 JDBC 接口的实现。

然后，我们通过提供连接 URL、用户名和密码来建立连接。

💡 数据库 URL 是指向您的数据库的地址。在我们的示例中，`localhost` 是主机名，`demoDB` 是数据库名称。不同的 JDBC 驱动有不同的对应 URL。

之后，我们创建一个 `Statement` 类型的对象来执行 SQL 语句，并返回结果集。

Spring MVC 中的 JDBC 连接

了解如何在 Java 中创建 JDBC 连接的基本概念对于在 Spring 中构建连接到数据库的 Web 应用程序至关重要。

但是，首先，让我们看看如何构建我们的应用程序。

- 结构

- **数据访问层 (DAO)**：通常用于数据库交互。他们应该通过一个接口公开这个功能，应用程序的其余部分将通过该接口访问它们。Spring 中的 DAO 支持使得使用数据访问技术（如 JDBC、Hibernate 等）变得容易。
- **JDBC 模板**：JDBC 模板执行 SQL 查询、更新语句、存储过程等。它还捕获 JDBC 异常。
- **映射器**：它们映射表和 Java 对象的行数据之间。在 ORM 的情况下，如 Hibernate，映射使用 `@Column` 对象的字段（稍后将讨论）。
- **实体**：它们是数据库表的表示。表中的每条记录代表一个实体对象。因此，实体对象的字段表示表列。

— JDBC 模板和 DAO



JDBC 依次实现 DAO 接口以提供实际的实现代码。虽然不是必须的，但可以直接在 DAO 类中编写实现代码，但这不是一个好习惯。

— JDBC 模板和数据源（即连接）

使用 JDBC 模板时的常见做法是：

1. 配置数据源在 Spring 配置 Bean。
2. 依赖注入类中的 *DataSource* bean 实现 DAO。
3. 在 *DataSource* 的 setter 中创建 *JdbcTemplate* 的新实例。

JDBC——过程

[1] 创建实体及其映射器

```
1  package com.springmvcdemo;
2
3  public class User {
4      private Integer id;
5      private String name;
6      private String email;
7
8      public User(){
9      }
10
11     /* getters and setters for objects' fields */
```



```
12 }
```

User.java hosted with ❤ by GitHub

[view raw](#)

```
1 package com.springmvcdemo;
2
3 import java.sql.ResultSet;
4 import java.sql.SQLException;
5 import org.springframework.jdbc.core.RowMapper;
6
7 public class UserMapper implements RowMapper<User> {
8
9     @Override
10    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
11        User user = new User();
12        user.setId(rs.getInt("id"));
13        user.setName(rs.getString("name"));
14        user.setEmail(rs.getString("email"));
15        return user;
16    }
17 }
```



UserMapper.java hosted with ❤ by GitHub

[view raw](#)

[2] 创建 DAO 接口

列出数据库交互的所有方法。

```
1 package com.springmvcdemo;
2
3 import java.util.List;
4 import javax.sql.DataSource;
5
6 public interface UserDao {
7
8    // set database connection
9    public void setDataSource(DataSource ds);
10
11    // create a new user record in the users table
12    public void create(String name, String email);
13
14    // get a user with the passed id
15    public User getUser(Integer id);
16
17    // get all the users from the users table
18    public List<User> listUsers();
19
20    // update a user's email given given the user's id
21    public void updateEmail(Integer id, String email);
22 }
```

```
22
23 // delete a user record from the users table given the user's id
24 public void delete(Integer id);
25 }
```

UserDAO.java hosted with ❤ by GitHub

[view raw](#)

【3】创建JDBC实现类

接下来是 DAO 接口的实现类。

💡 JDBC 实现类有一个 `JdbcTemplate` 实例，它通过使用 `DataSource` 对象（作为依赖项）来实例化来完成工作。

💡 您可以使用 XML 配置（Setter 注入）或使用注解（字段注入）注入 `DataSource` 对象。

```
1 package com.springmvcdemo;
2
3 import java.util.List;
4 import javax.sql.DataSource;
5
6 import org.springframework.jdbc.core.JdbcTemplate;
7
8 public class UserDAOJDBCImpl implements UserDAO {
9
10     private DataSource dataSource;
11     private JdbcTemplate jdbcTemplate;
12
13     @Override
14     public void setDataSource(DataSource ds) {
15         dataSource = ds;
16         jdbcTemplate = new JdbcTemplate(dataSource);
17     }
18
19     @Override
20     public void create(String name, String email) {
21         String sql = "insert into users (name, email) values (?, ?)";
22         jdbcTemplate.update(sql, name, email);
23         System.out.println("Created User Name = " + name + " Email = " + email);
24     }
25
26     @Override
27     public User getUser(Integer id) {
28         String sql = "select * from users where id = ?";
29         User user = jdbcTemplate.queryForObject(sql, new Object[]{id}, new UserMapper());
30         return user;
31     }
32 }
```



```

31     }
32
33     @Override
34     public List<User> listUsers() {
35         String sql = "select * from users";
36         List<User> users = jdbcTemplate.query(sql, new UserMapper());
37         return users;
38     }
39
40     @Override
41     public void updateEmail(Integer id, String email){
42         String sql = "update users set email = ? where id = ?";
43         jdbcTemplate.update(sql, id, email);
44         System.out.println("Updated Record with ID = " + id );
45     }
46
47     @Override
48     public void delete(Integer id){
49         String sql = "delete from users where id = ?";
50         jdbcTemplate.update(sql, id);
51         System.out.println("Deleted Record with ID = " + id );
52     }
53 }

```



💡使用Annotation 注入 DataSource 的情况。 @Autowired 在 jdbcTemplate 字段上使用注释，并删除该 setDataSource() 方法（也来自 UserDao）。

DataSource bean 将被注入到XML 中的JDBC 模板bean（见下文）。

[4] 定义配置

为DataSource和 JDBC 实现类定义配置。


```

1  <!-- Initialization for data source -->
2  <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
3      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
4      <property name="url" value="jdbc:mysql://localhost/demoDB"/>
5      <property name="username" value="root"/>
6      <property name="password" value=""/>
7  </bean>
8
9  <!-- Definition for UserDaoJDBCImpl bean (JDBC Implementation Class) -->
10 <bean id="UserDaoJDBCImpl" class="com.springmvcdemo.UserDaoJDBCImpl">
11     <property name="dataSource" ref="dataSource" />
12 </bean>

```

💡 使用 Annotation 注入 DataSource 的情况。

```
1  <!-- Initialization for data source -->
2  <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
3      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
4      <property name="url" value="jdbc:mysql://localhost/demoDB"/>
5      <property name="username" value="root"/>
6      <property name="password" value=""/>
7  </bean>
8
9  <!-- JDBC Template and Injecting dataSource -->
10 <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
11     <property name="dataSource" ref="dataSource"/></property>
12 </bean>
13
14 <!-- Definition for UserDaoJDBCImpl bean (JDBC Implementation Class) -->
15 <bean id="UserDAOJDBCImpl" class="com.springmvcdemo.UserDAOJDBCImpl" />
```

beans.xml hosted with  by GitHub[view raw](#)

[5] 在控制器中使用 JDBC 实现类

为了在我们的控制器中使用 `UserDAOJDBCImpl` 类，我们需要将它作为一个可以自动装配的依赖项注入到我们的控制器中。

△ `UserDAOJDBCImpl` 类必须定义为 bean 才能被注入。

然后，我们可以使用它来调用将与数据库交互并返回结果的方法。

最后，您可以根据返回的数据构造模型对象，以便在视图页面中显示。

```
1  package com.springmvcdemo;
2
3  import java.util.List;
4
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.stereotype.Controller;
7  import org.springframework.ui.Model;
8  import org.springframework.web.bind.annotation.ModelAttribute;
9  import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RequestMethod;
11 import org.springframework.web.bind.annotation.RequestParam;
12
13 @Controller
```




```
14 @RequestMapping("/user")
15 public class UserController {
16     @Autowired
17     private UserDAOJDBCImpl userJDBC;
18
19     @RequestMapping(value = "/list", method = RequestMethod.GET)
20     public String list(Model model){
21
22         // get all users
23         List<User> users = userJDBC.listUsers();
24
25         // add the users to the model
26         model.addAttribute("users", users);
27         return "list-users";
28     }
29
30     @RequestMapping(value = "/newUser", method = RequestMethod.GET)
31     public String newUser(Model model){
32         model.addAttribute("user", new User());
33         return "user-form";
34     }
35
36     @RequestMapping(value = "/addUser", method = RequestMethod.POST)
37     public String addUser(@ModelAttribute("user") User user){
38         String name      = user.getName();
39         String email     = user.getEmail();
40
41         // create a new user
42         userJDBC.create(name, email);
43
44         return "redirect:/user/list";
45     }
46
47     @RequestMapping(value = "/viewUser", method = RequestMethod.GET)
48     public String viewUser(@RequestParam("id") int id, Model model){
49         // get the user given the user's id
50         User user = userJDBC.getUser(id);
51
52         // add user as a model attribute to pre-populate the form
53         model.addAttribute("user", user);
54
55         return "user-form";
56     }
57
58     @RequestMapping(value = "/deleteUser", method = RequestMethod.GET)
59     public String deleteUser(@RequestParam("id") int id, Model model){
60         // delete a user
61         userJDBC.delete(id);
```

```
62
63     return "redirect:/user/list";
64 }
65 }
```

[6] 查看页面

一页显示所有用户，另一页使用 HTML 表单创建新用户。

```
1 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2
3 <html>
4 <head>
5     <title>Spring MVC Demo</title>
6 </head>
7 <body>
8     <h2>List Users</h2>
9     <ul>
10         <c:forEach var="user" items="${users}">
11             <li> ${user.id}, ${user.name}, ${user.email} </li>
12         </c:forEach>
13     </ul>
14 </body>
15 </html>
```

list-users.jsp hosted with ❤ by GitHub

[view raw](#)

```
1 <%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
2
3 <html>
4 <head>
5     <title>Spring MVC Demo</title>
6 </head>
7 <body>
8     <h2>User Data</h2>
9     <form:form action="addUser" modelAttribute="user" method="POST">
10
11         <form:label path="name">Name</form:label>
12         <form:input path="name" />
13
14         <form:label path="email">Email</form:label>
15         <form:input path="email" />
16
17         <input type="submit" value="Submit"/>
18     </form:form>
19 </body>
20 </html>
```



休眠

Hibernate 是一个在 java 对象和数据库记录之间进行映射的 ORM 框架。它充当我们的应用程序代码和数据库之间的中间层。

它处理低级 SQL 代码，最大限度地减少 JDBC 代码量，并提供对象到关系映射 (ORM)。实际上，Hibernate 在后台使用 JDBC 进行数据库通信。所以，它只是 JDBC 之上的一个抽象层。

我们要做的就是告诉 Hibernate 我们的 java 对象如何映射到数据库表。这是在对象类中（或在 XML 配置中）完成的。Hibernate 将根据我们定义的映射在表中获取、存储、更新或删除这个对象。

一个关于 Hibernate 如何让它变得更容易的简单例子.....



// 保存一个对象

```
User newUser = new User("John", "Doe", "myemail.com");
```

```
// "session" 是一个特殊的 Hibernate 对象
```

```
int thePrimaryId = (Integer) session.save(newUser);
```

// 检索一个对象

```
User user = session.get(User.class, thePrimaryId);
```

// 查询对象

```
// 我们传递所谓的“Hibernate 查询语言”（稍后将讨论）
```

```
Query query = session.createQuery("from user");
```

```
List<User> usersList = query.list();
```

Hibernate——过程

[1] Hibernate 配置文件

它告诉 hibernate 如何连接数据库，也就是 JDBC 配置，因为它使用 JDBC 连接数据库。因此，在“src”文件夹中创建一个“hibernate.cfg.xml”文件。

```
1 <!DOCTYPE hibernate-configuration PUBLIC
2     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
4
5 <hibernate-configuration>
6     <session-factory>
7         <!-- JDBC Database connection settings -->
```

```
8      <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
9      <property name="connection.url">jdbc:mysql://localhost/demoDB</property>
10     <property name="connection.username">root</property>
11     <property name="connection.password"></property>
12
13     <!-- JDBC connection pool settings ... using built-in test pool -->
14     <property name="connection.pool_size">1</property>
15
16     <!-- Select our SQL dialect -->
17     <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
18
19     <!-- Echo the SQL to stdout -->
20     <property name="show_sql">true</property>
21
22     <!-- Set the current session context -->
23     <property name="current_session_context_class">thread</property>
24
25 </session-factory>
26 </hibernate-configuration>
```



hibernate.cfg.xml hosted with ❤ by GitHub

[view raw](#)

该文件具有以下内容:

[1] Session Factory

它读取hibernate配置文件, 获取到数据库的连接, 并创建会话对象, 它在应用程序期间创建一次。

💡 会话是JDBC 连接的包装器。它是将对象保存和检索到数据库的主要对象。

它是短暂的对象, 因此, 对于给定的方法, 您使用会话, 然后将其丢弃, 依此类推。它是从会话工厂创建和检索的。

[2] JDBC 数据库连接设置: 这些是JDBC 连接的设置, 如JDBC 驱动程序、连接URL、用户名和密码。

[3] 连接池: 为简单起见, 暂时将其保留为 1。它是连接池中的最大连接数。

💡 Hibernate 利用连接池来连接数据库。

Hibernate 提供了一个基本的内部连接管理器。为了获得最佳性能和稳定性, 请使用第三方工具 (稍后会使用)。

[4] **SQL 方言**：它是您的数据库使用的 SQL 方言。因此，它告诉 Hibernate 为所选数据库生成适当的 SQL 语句。

[5] **显示 SQL**：打印出用于与数据库通信的 SQL。

[6] **会话上下文**：默认上下文是 `thread` 会话工厂将会话绑定到 `openSession()` 被调用的线程。

这很有用，因为您可以稍后调用 `sessionFactory.getCurrentSession()` 它将返回绑定到当前正在运行的线程的会话。

[2] 注释 Java 类（实体）

实体是映射到数据库表的普通 Java 类。我们使用注解来进行映射。这就是所谓的对象关系映射 (ORM)。

💡 还有另一种使用 XML 配置文件定义映射的方法，这是旧的方法。



所以，**首先**，将一个类映射到数据库表。**然后**，将字段映射到数据库表列。

```
1 package com.springmvcdemo;
2
3 import javax.persistence.Column;
4 import javax.persistence.Entity;
5 import javax.persistence.Id;
6 import javax.persistence.Table;
7
8 @Entity
9 @Table(name="users") // The default table name is the class name
10 public class User {
11
12     @Id // Identifies a primary key.
13     @Column(name="id") // The default name is the name of the field
14     private Integer id;
15
16     @Column(name="name")
17     private String name;
18
19     @Column(name="email")
20     private String email;
21
22     public User(){
23     }
24
25     public User(Integer id, String name, String email) {
```



```
26     this.id = id;
27     this.name = name;
28     this.email = email;
29 }
30
31 /* getters and setters for objects' fields */
32 }
```

User.java hosted with ❤ by GitHub

[view raw](#)

⚠不要忘记定义构造函数、getter 和 setter。

[3] 设置会话工厂并创建会话对象

会话工厂创建一个会话对象，该对象又将用于与数据库进行交互。

首先，我们加载配置文件，默认名称配置文件是“hibernate.cfg.xml”。然后，要扫描的类进行映射，然后我们创建会话对象。

最后，关闭工厂会话。它销毁SessionFactory并释放所有资源（缓存、连接池等）



```
1  import org.hibernate.Session;
2  import org.hibernate.SessionFactory;
3  import org.hibernate.cfg.Configuration;
4
5  public class Test {
6      public static void main(String[] args) {
7          SessionFactory factory = new Configuration()
8              .configure("hibernate.cfg.xml")
9              .addAnnotatedClass(User.class)
10             .buildSessionFactory();
11
12         Session session = factory.getCurrentSession();
13         try {
14             // your database code
15         } catch (Exception e){
16
17         } finally {
18             factory.close();
19         }
20     }
21 }
```

Test.java hosted with ❤ by GitHub

[view raw](#)

[4] 开始交易

在 Hibernate 中, 与数据库交互时始终使用数据库事务。无论是创建、读取、更新还是删除操作。在 Hibernate 中, 数据库事务从不是可选的。

// 开始一个事务

```
session.beginTransaction();
```

// 数据库操作

// ... 我们可以在事务中包装一组查询

// 提交事务

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

该 `commit()` 语句指示事务结束, 并将更改提交到数据库。除非提交事务, 否则不会对数据库应用任何更改。

一旦提交事务, 就不能再使用会话对象。您需要重新分配会话对象以获取新会话再次开始事务。



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

[4] 保存 (创建) 对象到数据库

对于每个操作, 不要忘记使用 获取一个新的会话对象 `factory.getCurrentSession()`, 启动事务并提交。

// 创建一个用户对象

```
User newUser = new User(125, "Alex", "alex@example.com");
```

// 保存用户对象

```
session.save(newUser);
```

[5] 读取对象

// 根据id检索学生: primary key

```
User user = session.get(User.class, newUser.getId());
```

Hibernate 中有一种称为HQL的查询语言, 用于查询数据库。在 HQL 中, 我们使用类名和属性名, 而不是表名和列名。

```
// "User" 是类
List<User> users = session.createQuery("from User").list();

// "u" 是映射到实际 java 对象（用户）的别名，
// 其中 u.name 是用户对象的属性
List<User> users = session.createQuery("from User AS u where u.name
='亚历克斯'").list();
```

△在Hibernate 5.2 或更高版本中，Query `list()` 方法已被弃用，请 `getResultList()` 改用。

[6] 更新对象

我们可以使用setters更新用户对象。

💡由于用户对象的持久实例是通过 `get()` 方法返回的（从表中获取后），更新此用户对象的字段值也会反映在表值上。



```
// 根据 id 检索用户：主键
int userId = 125;
User user = session.get(User.class, userId);

// 更新名称为 "Max"
user.setName("Max");
```

或者，我们可以编写查询来更新特定用户。

```
session.createQuery("更新用户集名称='Max' WHERE
id=125").executeUpdate();
```

[7] 删除对象

更新中的相同想法在这里也适用；我们可以删除 `get()` 方法返回的用户对象的持久实例，或者编写并执行查询来执行此操作。

```
// 根据 id 检索用户：主键
int userId = 125;
User user = session.get(User.class, userId);

// 删除学生
session.delete(user);
```

```
// 删除用户 id=125 (使用查询)
```

```
session.createQuery("delete from User WHERE  
id=125").executeUpdate();
```

感谢您的阅读！如果你喜欢它，请为它鼓掌👏。

爪哇 弹簧框架 编程 软件开发 其他



关于 写 帮助 合法的

获取 Medium 应用

