

Spring Boot 和 JdbcTemplate 介绍: JDBC 模板



奥塔维奥·桑塔纳

跟随

4月7日 · 4分钟阅读

与任何编程语言一样，Java 有多种工具可以在语言和数据库之间轻松集成。有多种工具，例如 Hibernate、Eclipse Link、JPA 规范等。但是，ORM 带来了几个问题，有时使用它没有意义，然后将其用作 Java 通信层或 JDBC。本教程将介绍一种使用 `Spring` JDBC 模板简化 JDBC 代码的方法。



ORM 等映射框架减少了大量样板文件，减少了重复代码，避免了错误，并且不会重新发明轮子。然而，当 RDBMS 由以面向对象的编程语言编写的应用程序提供服务时，经常会遇到对象-关系阻抗失配，这会带来一系列概念和技术困难。

解决方案可能使用 JDBC, 但它增加了处理数据和 Java 的复杂性。应用程序如何使用 JDBC 减少这种冗长性? Spring JDBCTemplate 是一种强大的连接数据库和执行 SQL 查询的机制。它内部使用 JDBC API, 但消除了 JDBC API 的很多问题。

从 Spring Initializr 开始

此示例应用程序将使用 JDBCTemplate 和 Spring 来使用两个数据库: PostgreSQL 来运行应用程序和 H2 在测试范围内。对于所有 Spring 应用程序, 您应该从 [Spring Initializr](#) 开始。Initializr 提供了一种快速的方法来拉入应用程序所需的所有依赖项, 并为您完成大量设置。此示例需要 JDBC API、Spring MVC、PostgreSQL 驱动程序和 H2 数据库依赖项。

```
1  <dependency>
2      <groupId>org.springframework.boot</groupId>
3      <artifactId>spring-boot-starter-web</artifactId>
4  </dependency>
5  <dependency>
6      <groupId>org.springframework.boot</groupId>
7      <artifactId>spring-boot-starter-jdbc</artifactId>
8  </dependency>
9  <dependency>
10     <groupId>org.postgresql</groupId>
11     <artifactId>postgresql</artifactId>
12     <scope>runtime</scope>
13 </dependency>
14 <dependency>
15     <groupId>com.h2database</groupId>
16     <artifactId>h2</artifactId>
17     <scope>test</scope>
18 </dependency>
19 <dependency>
20     <groupId>org.springframework.boot</groupId>
21     <artifactId>spring-boot-starter-test</artifactId>
22     <scope>test</scope>
23 </dependency>
```

pom.xml hosted with  by GitHub

[view raw](#)

下一步是 Car 实体, 有五个字段: id、name、city、model 和 color。在这个项目中, 作为贫血模型的 POJO 足以处理它。丰富的模型保证了对对象的规则并避免了任何封装问题, 因此是一个防弹 API。但是, 它只适用于复杂的项目。强调: 当我们谈论软件架构时, “这取决于”总是一个很好的答案。

```
1  public class Car {
```

```
2
3     private Long id;
4
5     private String name;
6
7     private String city;
8
9     private String model;
10
11    private String color;
12    //...
13    public static CarBuilder builder() {
14        return new CarBuilder();
15    }
16 }
```

Car.java hosted with ❤ by GitHub

[view raw](#)

为了在 Java 和 JDBC 的 ResultSet 之间进行紧密集成, Spring JDBC 有一个 RowMapper 接口。开发人员可以创建自定义类或使用 [BeanPropertyRowMapper](#), 减少了样板; 但是, 实体应该有一个公共的 getter 和 setter; 除了提供便利而不是高性能之外, 它可能会遇到封装问题。为了获得最佳性能, 请考虑使用自定义 [RowMapper](#) 实现。



```
1     public class CarRowMapper implements RowMapper<Car> {
2
3         @Override
4         public Car mapRow(ResultSet resultSet, int rowNum) throws SQLException {
5             Long id = resultSet.getLong("ID");
6             String name = resultSet.getString("name");
7             String city = resultSet.getString("city");
8             String model = resultSet.getString("model");
9             String color = resultSet.getString("color");
10            return Car.builder().id(id).name(name)
11                .city(city)
12                .model(model)
13                .color(color).build();
14        }
15    }
```

CarRowMapper.java hosted with ❤ by GitHub

[view raw](#)

实体已准备就绪; 我们来谈谈数据访问对象, DAO; 每次有一个庞大或复杂的查询需要处理时, 都会讨论 SQL 查询应该去哪里。简而言之, 当脚本被硬编码时, 脚本在做

什么就更清楚了，但是当命令非常庞大时，阅读和理解就变得具有挑战性。因此，我们可以移动它以从属性中读取它。

```
1 car.query.find.by.id=SELECT * FROM CAR WHERE ID = :id
2 car.query.delete.by.id=DELETE FROM CAR WHERE ID =:id
3 car.query.update=update CAR set name = :name, city = :city, model= :model, color =:color where
4 car.query.find.all=select * from CAR ORDER BY id LIMIT :limit OFFSET :offset
```

application.properties hosted with ❤ by GitHub

[view raw](#)

一旦读者熟悉了代码中的查询，我们将探索属性文件中的查询选项。我们将有一个类来承担这个职责，它将与 Spring Configuration 紧密集成。

```
1 @Component
2 public class CarQueries {
3
4     @Value("${car.query.find.by.id}")
5     private String findById;
6     @Value("${car.query.delete.by.id}")
7     private String deleteById;
8     @Value("${car.query.update}")
9     private String update;
10    @Value("${car.query.find.all}")
11    private String findAll;
12    //...
13 }
```

CarQueries.java hosted with ❤ by GitHub

[view raw](#)

一旦 findAll 具有分页支持，CarDAO 将执行 CRUD 操作。它使用 NamedParameterJdbcTemplate 对一组基本的 JDBC 操作进行分类，允许命名参数而不是传统的“?” 占位符。为了避免连接泄漏，Spring 有 Transactional 注释来控制我们在代码中定义的每个方法中的事务。

```
1 @Repository
2 public class CarDAO {
3
4     private final NamedParameterJdbcTemplate template;
5     private final CarQueries queries;
6     private final RowMapper<Car> rowMapper;
7     private final SimpleJdbcInsert insert;
8
9     @Autowired
10    public CarDAO(NamedParameterJdbcTemplate template, CarQueries queries) {
```



```

10 public CarDAO(NamedParameterJdbcTemplate template, CarQueries queries) {
11     this.template = template;
12     //this.rowMapper = new BeanPropertyRowMapper<>(Car.class);
13     this.rowMapper = new CarRowMapper();
14     this.queries = queries;
15     this.insert = new SimpleJdbcInsert(template.getJdbcTemplate());
16     this.insert.setTableName("car");
17     this.insert.usingGeneratedKeyColumns("id");
18 }
19
20 @Transactional
21 public Car insert(Car car) {
22     //Number id = insert.executeAndReturnKey(new BeanPropertySqlParameterSource(car));
23     Number id = insert.executeAndReturnKey(car.toMap());
24     return findById(id.longValue()).orElseThrow(() -> new IllegalStateException(""));
25 }
26
27 public Optional<Car> findById(Long id) {
28     String sql = queries.getFindById();
29     Map<String, Object> parameters = Collections.singletonMap("id", id);
30     return template.queryForStream(sql, parameters, rowMapper).findFirst();
31 }
32
33 @Transactional
34 public boolean delete(Long id) {
35     String sql = queries.getDeleteById();
36     Map<String, Object> paramMap = Collections.singletonMap("id", id);
37     return template.update(sql, paramMap) == 1;
38 }
39
40 @Transactional
41 public boolean update(Car car) {
42     String sql = queries.getUpdate();
43     Map<String, Object> paramMap = car.toMap();
44     return template.update(sql, paramMap) == 1;
45 }
46
47 public Stream<Car> findAll(Page page) {
48     String sql = queries.getFindAll();
49     Map<String, Object> paramMap = new HashMap<>();
50     paramMap.put("limit", page.getLimit());
51     paramMap.put("offset", page.getOffset());
52     return template.queryForStream(sql, paramMap, rowMapper);
53 }
54 }

```



代码准备好了; 让我们测试一下。是的, TDD 技术有一个哲学, 即从测试开始, 然后创建代码。但这不是本文的目的。在测试范围内, 我们将使用 H2 在内存中生成一个 DBMS。Spring 有几个特性可以让我们顺利进行测试。感谢 Spring, 我们可以在测试中使用 H2, 而不会影响将在生产中运行的驱动程序。

```
1  @ExtendWith(SpringExtension.class)
2  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
3  class CarDAOTest {
4
5      @Autowired
6      private CarDAO carDAO;
7
8      @Autowired
9      private JdbcTemplate template;
10
11     @Test
12     public void shouldFindById() {
13         Assertions.assertNotNull(carDAO);
14         Optional<Car> car = carDAO.findById(1L);
15         Assertions.assertNotNull(car);
16     }
17
18     @Test
19     public void shouldInsertCar() {
20         Car car = Car.builder()
21             .city("Salvador")
22             .color("Red")
23             .name("Fiat")
24             .model("Model")
25             .build();
26         Car insert = carDAO.insert(car);
27         Assertions.assertNotNull(insert);
28         Assertions.assertNotNull(insert.getId());
29     }
30
31     @Test
32     public void shouldDelete() {
33         Car car = Car.builder()
34             .city("Salvador")
35             .color("Red")
36             .name("Fiat")
37             .model("Model")
38             .build();
39         Car insert = carDAO.insert(car);
40         carDAO.delete(insert.getId());
41         Optional<Car> empty = carDAO.findById(insert.getId());
```





```
42     Assertions.assertTrue(empty.isEmpty());
43 }
44
45 @Test
46 public void shouldUpdate() {
47     Car car = Car.builder()
48         .city("Salvador")
49         .color("Red")
50         .name("Fiat")
51         .model("Model")
52         .build();
53     Car insert = carDAO.insert(car);
54
55     insert.update(Car.builder()
56         .city("Salvador")
57         .color("Red")
58         .name("Fiat")
59         .model("Update")
60         .build());
61     carDAO.update(insert);
62 }
63
64 @Test
65 public void shouldFindAll() {
66     template.execute("DELETE FROM CAR");
67     List<Car> cars = new ArrayList<>();
68     for (int index = 0; index < 10; index++) {
69         Car car = Car.builder()
70             .city("Salvador")
71             .color("Red")
72             .name("Fiat " + index)
73             .model("Model")
74             .build();
75         cars.add(carDAO.insert(car));
76     }
77     Page page = Page.of(1, 2);
78     List<Car> result = carDAO.findAll(page).collect(Collectors.toList());
79     Assertions.assertEquals(2, result.size());
80     MatcherAssert.assertThat(result, Matchers.contains(cars.get(0), cars.get(1)));
81     Page nextPage = page.next();
82     result = carDAO.findAll(nextPage).collect(Collectors.toList());
83     Assertions.assertEquals(2, result.size());
84     MatcherAssert.assertThat(result, Matchers.contains(cars.get(2), cars.get(3)));
85 }
86 }
```

在本教程中，我们介绍了 Spring JDBC 及其操作；除了测试和配置资源之外，我们还讨论了映射权衡和存储查询的位置。Spring 带来了几个提高开发人员生产力的特性。在第二部分，我们将讨论一些关于 Spring MVC 及其与数据库的工作。

代码： <https://github.com/xgeekshq/spring-boot-jdbc-template-sample>

如果您喜欢在具有全球影响力的项目中进行大规模工作，并且喜欢 Java 的真正挑战，请随时通过 [xgeeks](#) 与我们联系！我们正在壮大我们的团队，尤其是在 Java 方面，您可能是下一个加入这群才华横溢的人 😊

如果您想一睹 xgeeks 的生活，请查看我们的社交媒体频道！再见！

爪哇 春天 弹簧靴 数据库 休息



[关于](#) [写](#) [帮助](#) [合法的](#)

获取 Medium 应用

