

13 | JDBC访问框架：如何抽取JDBC模板并隔离数据库？

2023-04-10 郭屹 来自北京

《手把手带你写一个MiniSpring》



你好，我是郭屹，今天我们继续手写 MiniSpring。从这节课开始我们进入 MiniSpring 一个全新的部分：JdbcTemplate。

到现在为止，我们的 MiniSpring 已经成了一个相对完整的简易容器，具备了基本的 IoC 和 MVC 功能。现在我们就在这个简易容器的基础之上，继续添加新的特性。首先就是**数据访问的特性**，这是任何一个应用系统的基本功能，所以我们先实现它。这之后，我们的 MiniSpring 就基本落地了，你真的可以以它为框架进行编程了。


我们还是先从标准的 JDBC 程序开始探讨。

JDBC 通用流程

在 Java 体系中，数据访问的规范是 JDBC，也就是 Java Database Connectivity，想必你已经熟悉或者至少听说过，一个简单而典型的 JDBC 程序大致流程是怎样的呢？我们一步步来

看，每一步我也会给你放上一两个代码示例帮助你理解。

第一步，加载数据库驱动程序。

 复制代码


```
1 Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

或者直接 `new Driver();` 也可以。

这是第一步，因为 JDBC 只是提供了一个访问的 API，具体访问数据库的工作是由不同厂商提供的数据库 driver 来实现的，Java 只是规定了这个通用流程。对同一种数据库，可以有不同的 driver，我们也可以自己按照协议实现一个 driver，我自己就曾在 1996 年实现了中国第一个 JDBC Driver。

这里我多提一句，Java 的这种设计很是巧妙，让应用程序的 API 与对应厂商的 SPI 分隔开了，它们可以各自独立进化，这是通过一种叫“桥接模式”的办法达到的。这节课你就能切身感受到这种模式的应用效果了。

第二步，获取数据库连接。


 复制代码

```
1 con = DriverManager.getConnection("jdbc:sqlserver://localhost:1433;databasename
```


`getConnection()` 方法的几个参数，分别表示数据库 URL、登录数据库的用户名和密码。

这个时候，我们利用底层 driver 的功能建立了对数据库的连接。不过要注意了，建立和断开连接的过程是很费时间的，所以后面我们会利用数据库连接池技术来提高性能。

第三步，通过 `Connection` 对象创建 `Statement` 对象，比如下面这两条。

 复制代码

```
1 stmt = con.createStatement(sql);
```

 复制代码

```
1 stmt = con.prepareStatement(sql);
```

Statement 是对一条 SQL 命令的包装。

第四步，使用 Statement 执行 SQL 语句，还可以获取返回的结果集 ResultSet。

```
1 rs = stmt.executeQuery();
```

 复制代码

```
1 stmt.executeUpdate();
```

 复制代码

第五步，操作 ResultSet 结果集，形成业务对象，执行业务逻辑。

```
1 User rtnUser = null;
2 if (rs.next()) {
3     rtnUser = new User();
4     rtnUser.setId(rs.getInt("id"));
5     rtnUser.setName(rs.getString("name"));
6 }
```

 复制代码

第六步，回收数据库资源，关闭数据库连接，释放资源。

```
1 rs.close();
2 stmt.close();
3 con.close();
```


 复制代码

这个数据访问的套路或者定式，初学 Java 的程序员都比较熟悉。写多了 JDBC 程序，我们会发现 Java 里面访问数据的程序结构都是类似的，不一样的只是具体的 SQL 语句，然后还有一点就是执行完 SQL 语句之后，每个业务对结果的处理是不同的。只要稍微用心思考一下，你就会想到应该把它做成一个模板，方便之后使用，自然会去抽取 JdbcTemplate。

抽取 JdbcTemplate

抽取的基本思路是**动静分离，将固定的套路作为模板定下来，变化的部分让子类重写**。这是常用的设计模式，基于这个思路，我们考虑提供一个 JdbcTemplate 抽象类，实现基本的 JDBC 访问框架。

以数据查询为例，我们可以在这个框架中，让应用程序员传入具体要执行的 SQL 语句，并把返回值的处理逻辑设计成一个模板方法让应用程序员去具体实现。

 复制代码

```
1 package com.minis.jdbc.core;
2
3 import java.sql.Connection;
4 import java.sql.DriverManager;
5 import java.sql.PreparedStatement;
6 import java.sql.ResultSet;
7
8 public abstract class JdbcTemplate {
9     public JdbcTemplate() {
10    }
11    public Object query(String sql) {
12        Connection con = null;
13        PreparedStatement stmt = null;
14        ResultSet rs = null;
15        Object rtnObj = null;
16
17        try {
18            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
19            con = DriverManager.getConnection("jdbc:sqlserver://localhost:1433;database
20
21            stmt = con.prepareStatement(sql);
22            rs = stmt.executeQuery();
23
24            //调用返回数据处理方法，由程序员自行实现
25            rtnObj = doInStatement(rs);
26        }
27        catch (Exception e) {
```


```

28         e.printStackTrace();
29     }
30     finally {
31         try {
32             rs.close();
33             stmt.close();
34             con.close();
35         } catch (Exception e) {
36         }
37     }
38     return rtnObj;
39 }
40
41 protected abstract Object doInStatement(ResultSet rs);
42 }
43

```

通过上述代码我们可以看到，query() 里面的代码都是模式化的，SQL 语句作为参数传进来，最后处理 SQL 返回数据的业务代码，留给应用程序员自己实现，就是这个模板方法 doInStatement()。这样就实现了动静分离。

比如说，我们数据库里有一个数据表 User，程序员可以用一个数据访问类 UserJdbcImpl 进行数据访问，你可以看一下代码。

 复制代码

```

1 package com.test.service;
2
3 import java.sql.ResultSet;
4 import java.sql.SQLException;
5 import com.minis.jdbc.core.JdbcTemplate;
6 import com.test.entity.User;
7
8 public class UserJdbcImpl extends JdbcTemplate {
9     @Override
10    protected Object doInStatement(ResultSet rs) {
11        //从jdbc数据集读取数据，并生成对象返回
12        User rtnUser = null;
13        try {
14            if (rs.next()) {
15                rtnUser = new User();
16                rtnUser.setId(rs.getInt("id"));
17                rtnUser.setName(rs.getString("name"));
18                rtnUser.setBirthday(new java.util.Date(rs.getDate("birthday").getTime()))

```

```
19     } else {
20     }
21 } catch (SQLException e) {
22     e.printStackTrace();
23 }
24
25     return rtnUser;
26 }
27 }
```

应用程序员在自己实现的 `doInStatement()` 里获得 SQL 语句的返回数据集并进行业务处理，返回一个业务对象给用户类。

而对外提供服务的 `UserService` 用户类就可以简化成下面这样。

 复制代码

```
1 package com.test.service;
2
3 import com.minis.jdbc.core.JdbcTemplate;
4 import com.test.entity.User;
5
6 public class UserService {
7     public User getUserInfo(int userid) {
8         String sql = "select id, name,birthday from users where id="+userid;
9         JdbcTemplate jdbcTemplate = new UserJdbcImpl();
10        User rtnUser = (User)jdbcTemplate.query(sql);
11
12        return rtnUser;
13    }
14 }
```

我们看到，用户类简单地创建一个 `UserJdbcImpl` 对象，然后执行 `query()` 即可，很简单。

有了这个简单的模板，我们就做到了把 JDBC 程序流程固化下来，分离出变化的部分，让应用程序员只需要管理 SQL 语句并处理返回的数据就可以了。


这是一个实用的结构，我们就基于这个结构继续往前走。

通过 Callback 模式简化业务实现类

上面抽取出来的 Tempalte，我们也看到了，如果只是停留在现在的这一步，那应用程序的工作量还是很大的，对每一个数据表的访问都要求手写一个对应的 JdbcImpl 实现子类，很繁琐。为了不让每个实体类都手写一个类似于 UserJdbcImpl 的类，我们可以采用 Callback 模式来达到目的。


先介绍一下 Callback 模式，它是把一个需要被调用的函数作为一个参数传给调用函数。你可以看一下基本的做法。

先定义一个回调接口。

 复制代码

```
1 public interface Callback {  
2     void call();  
3 }
```


有了这个 Callback 接口，任务类中可以把它作为参数，比如下面的业务任务代码。

 复制代码

```
1 public class Task {  
2     public void executeWithCallback(Callback callback) {  
3         execute(); //具体的业务逻辑处理  
4         if (callback != null) callback.call();  
5     }  
6 }
```

这个任务类会先执行具体的业务逻辑，然后调用 Callback 的回调方法。


用户程序如何使用它呢？

 复制代码

```
1     public static void main(String[] args) {  
2         Task task = new Task();  
3         Callback callback = new Callback() {
```

```
4         public void call() {
5             System.out.println("callback...");
6         }
7     };
8     task.executeWithCallback(callback);
9 }
```

先创建一个任务类，然后定义具体的回调方法，最后执行任务的同时将 Callback 作为参数传进去。这里可以看到，回调接口是一个单一方法的接口，我们可以采用函数式编程进一步简化它。

 复制代码

```
1     public static void main(String[] args) {
2         Task task = new Task();
3         task.executeWithCallback(()->{System.out.println("callback;")});
4     }
```

上面就是 Callback 模式的实现，我们把一个回调函数作为参数传给了调用者，调用者在执行完自己的任务后调用这个回调函数。

现在我们就按照这个模式改写 JdbcTemplate 的 query() 方法。

 复制代码

```
1     public Object query(StatementCallback stmtcallback) {
2         Connection con = null;
3         Statement stmt = null;
4
5         try {
6             Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
7             con = DriverManager.getConnection("jdbc:sqlserver://localhost:1433;database
8
9             stmt = con.createStatement();
10
11             return stmtcallback.doInStatement(stmt);
12         }
13         catch (Exception e) {
14             e.printStackTrace();
15         }
16         finally {
```



```


17         try {
18             stmt.close();
19             con.close();
20         } catch (Exception e) {
21             }
22     }
23     return null;
24 }

```

从代码中可以看出，在 query() 方法中增加了一个参数：StatementCallback，这就是需要回调的方法。这里我还要提醒你一下，Java 是纯粹的面向对象编程，没有真正的全局函数，所以实际代码中是一个类。

有了这个回调参数，就不需要给每一个数据访问增加一个子类来实现 doInStatement() 了，而是作为参数传进去。

你可以看一下 Callback 接口。

 复制代码

```

1 package com.minis.jdbc.core;
2
3 import java.sql.SQLException;
4 import java.sql.Statement;
5
6 public interface StatementCallback {
7     Object doInStatement(Statement stmt) throws SQLException;
8 }

```

可以看出这是一个函数式接口。

现在，应用程序就只需要用一个 JdbcTemplate 类就可以了，不用再为每一个业务类单独做一个子类。就像我们前面说的，用户类需要使用 Callback 动态匿名类的方式进行改造。

代码如下：

 复制代码

```

1 public User getUserInfo(int userid) {

```


```

2     final String sql = "select id, name,birthday from users where id="+userid;
3     return (User)jdbcTemplate.query(
4         (stmt)->{
5             ResultSet rs = stmt.executeQuery(sql);
6             User rtnUser = null;
7             if (rs.next()) {
8                 rtnUser = new User();
9                 rtnUser.setId(userid);
10                rtnUser.setName(rs.getString("name"));
11                rtnUser.setBirthday(new java.util.Date(rs.getDate("birthday").getTime
12            )
13            return rtnUser;
14        }
15    );
16 }

```

从代码中可以看到，以前写在 UserJdbcImpl 里的业务代码，也就是对 SQL 语句返回值的处理逻辑，现在成了匿名类，作为参数传入 query() 里，最后在 query() 里会回调到它。

按照同样的办法我们还可以支持 PreparedStatement 类型，方法调用时带上 SQL 语句需要的参数值。

 复制代码

```


1     public Object query(String sql, Object[] args, PreparedStatementCallback pstmtc
2         //省略获取connection等代码
3         pstmt = con.prepareStatement(sql);
4         for (int i = 0; i < args.length; i++) { //设置参数
5             Object arg = args[i];
6             //按照不同的数据类型调用JDBC的不同设置方法
7             if (arg instanceof String) {
8                 pstmt.setString(i+1, (String)arg);
9             } else if (arg instanceof Integer) {
10                pstmt.setInt(i+1, (int)arg);
11            }
12        }
13        return pstmtcallback.doInPreparedStatement(pstmt);
14    }

```

通过代码可以知道，和普通的 Statement 相比，这个 PreparedStatement 场景只是需要额外对 SQL 参数一个个赋值。这里我们还要注意一点，当 SQL 语句里有多个参数的时候，

MiniSpring 会按照参数次序赋值，和参数名没有关系。


我们再来看一下为 PreparedStatement 准备的 Callback 接口。

 复制代码

```
1 package com.minis.jdbc.core;
2
3 import java.sql.PreparedStatement;
4 import java.sql.SQLException;
5
6 public interface PreparedStatementCallback {
7     Object doInPreparedStatement(PreparedStatement stmt) throws SQLException;
8 }
```

这也是一个函数式接口。

用户服务类代码改造如下：

 复制代码


```
1 public User getUserInfo(int userid) {
2     final String sql = "select id, name,birthday from users where id=?";
3     return (User)jdbcTemplate.query(sql, new Object[]{new Integer(userid)},
4         (pstmt)->{
5             ResultSet rs = pstmt.executeQuery();
6             User rtnUser = null;
7             if (rs.next()) {
8                 rtnUser = new User();
9                 rtnUser.setId(userid);
10                rtnUser.setName(rs.getString("name"));
11            }
12            return rtnUser;
13        }
14    );
15 }
```

到这里，我们就用一个单一的 JdbcTemplate 类实现了数据访问。

结合 IoC 容器

当然，我们还可以更进一步，既然我们的 MiniSpring 是个 IoC 容器，可以管理一个一个的 Bean 对象，那么我们就好好利用它。由于只需要唯一的一个 JdbcTemplate 类，我们就可以事先把它定义为一个 Bean，放在 IoC 容器里，然后通过 @Autowired 自动注入。

在 XML 配置文件中声明一下。

 复制代码

```
1 <bean id="jdbcTemplate" class="com.minis.jdbc.core.JdbcTemplate" />
```

上层用户 service 程序中就不需要自己手动创建 JdbcTemplate，而是通过 Autowired 注解进行注入就能得到了。

 复制代码


```
1 package com.test.service;
2
3 import java.sql.ResultSet;
4 import com.minis.beans.factory.annotation.Autowired;
5 import com.minis.jdbc.core.JdbcTemplate;
6 import com.test.entity.User;
7
8 public class UserService {
9     @Autowired
10     JdbcTemplate jdbcTemplate;
11 }
```

我们需要记住，MiniSpring 只支持按照名字匹配注入，所以 UserService 类里的实例变量 JdbcTemplate 这个名字必须与 XML 文件中配置的 Bean 的 id 是一致的。如果不一致就会导致程序找不到 JdbcTemplate。

这样一来，应用程序中和数据库访问相关的代码就全部剥离出去了，应用程序只需要声明使用它，而它的创建、管理都由 MiniSpring 框架来完成。从这里我们也能看出 IoC 容器带来的便利，事实上，我们需要用到的很多工具，都会以 Bean 的方式在配置文件中声明，交给 IoC 容器来管理。

数据源

我们注意到，JdbcTemplate 中获取数据库连接信息等套路性语句仍然是硬编码的（hard coded）。

 复制代码


```
1 Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
2 con = DriverManager.getConnection("jdbc:sqlserver://localhost:1433;databasename=D
```

现在我们动手把这一部分代码包装成 DataSource，通过它获取数据库连接。假设有了这个工具，上层应用程序就简单了。你可以看一下使用者的代码示例。

 复制代码


```
1 con = dataSource.getConnection();
```

这个 Data Source 被 JdbcTemplate 使用。

 复制代码

```
1 public class JdbcTemplate {
2     private DataSource dataSource;
3 }
```

而这个属性可以通过配置注入，你可以看下配置文件。


 复制代码

```
1 <bean id="dataSource" class="com.minis.jdbc.datasource.SingleConnectionDataSource
2     <property type="String" name="driverClassName" value="com.microsoft.sqlserver.j
3     <property type="String" name="url" value="jdbc:sqlserver://localhost:1433;datab
4     <property type="String" name="username" value="sa"/>
5     <property type="String" name="password" value="Sql2016"/>
6 </bean>
7 <bean id="jdbcTemplate" class="com.minis.jdbc.core.JdbcTemplate" >
8     <property type="javax.sql.DataSource" name="dataSource" ref="dataSource"/>
9 </bean>
```

在 DataSource 这个 Bean 初始化的时候，设置 Property 时会加载相应的 JDBC Driver，然后注入给 JdbcTemplate 来使用。

我们再次看到，独立抽取这些部件，加上 IoC 容器的 Bean 管理，给系统构造带来许多便利。

上面描述的是假定有了一个 DataSource 之后怎么使用，现在回头再来看 DataSource 本身是怎么构造出来的。其实 Java 里已经给出了这个接口，是 javax.sql.DataSource。我们就遵守这个规范，做一个简单的实现。

 复制代码

```
1 package com.minis.jdbc.datasource;
2
3 import java.io.PrintWriter;
4 import java.sql.Connection;
5 import java.sql.DriverManager;
6 import java.sql.SQLException;
7 import java.sql.SQLFeatureNotSupportedException;
8 import java.util.Properties;
9 import java.util.logging.Logger;
10 import javax.sql.DataSource;
11
12 public class SingleConnectionDataSource implements DataSource {
13     private String driverClassName;
14     private String url;
15     private String username;
16     private String password;
17     private Properties connectionProperties;
18     private Connection connection;
19
20     //默认构造函数
21     public SingleConnectionDataSource() {
22     }
23     //一下是属性相关的getter和setter
24     public String getUrl() {
25         return url;
26     }
27     public void setUrl(String url) {
28         this.url = url;
29     }
30     public String getUsername() {
31         return username;
32     }
33     public void setUsername(String username) {
34         this.username = username;
```

```
35     }
36     public String getPassword() {
37         return password;
38     }
39     public void setPassword(String password) {
40         this.password = password;
41     }
42     public Properties getConnectionProperties() {
43         return connectionProperties;
44     }
45     public void setConnectionProperties(Properties connectionProperties) {
46         this.connectionProperties = connectionProperties;
47     }
48     @Override
49     public PrintWriter getLogWriter() throws SQLException {
50         return null;
51     }
52     @Override
53     public int getLoginTimeout() throws SQLException {
54         return 0;
55     }
56     @Override
57     public Logger getParentLogger() throws SQLFeatureNotSupportedException {
58         return null;
59     }
60     @Override
61     public void setLogWriter(PrintWriter arg0) throws SQLException {
62     }
63     @Override
64     public void setLoginTimeout(int arg0) throws SQLException {
65     }
66     @Override
67     public boolean isWrapperFor(Class<?> arg0) throws SQLException {
68         return false;
69     }
70     @Override
71     public <T> T unwrap(Class<T> arg0) throws SQLException {
72         return null;
73     }
74     //设置driver class name的方法, 要加载driver类
75     public void setDriverClassName(String driverClassName) {
76         this.driverClassName = driverClassName;
77         try {
78             Class.forName(this.driverClassName);
79         }
80         catch (ClassNotFoundException ex) {
81             throw new IllegalStateException("Could not load JDBC driver class [" + driv
82         }
83     }
```

```

84     //实际建立数据库连接
85     @Override
86     public Connection getConnection() throws SQLException {
87         return getConnectionFromDriver(getUsername(), getPassword());
88     }
89     @Override
90     public Connection getConnection(String username, String password) throws SQLException {
91         return getConnectionFromDriver(username, password);
92     }
93     //将参数组织成Properties结构, 然后拿到实际的数据库连接
94     protected Connection getConnectionFromDriver(String username, String password)
95         Properties mergedProps = new Properties();
96         Properties connProps = getConnectionProperties();
97         if (connProps != null) {
98             mergedProps.putAll(connProps);
99         }
100        if (username != null) {
101            mergedProps.setProperty("user", username);
102        }
103        if (password != null) {
104            mergedProps.setProperty("password", password);
105        }
106
107        this.connection = getConnectionFromDriverManager(getUrl(), mergedProps);
108        return this.connection;
109    }
110    //通过DriverManager.getConnection()建立实际的连接
111    protected Connection getConnectionFromDriverManager(String url, Properties props)
112        return DriverManager.getConnection(url, props);
113    }
114 }

```

这个类很简单, 封装了和数据访问有关的信息, 除了 getter 和 setter 之外, 它最核心的方法就是 `getConnection()`, 这个方法又会调用 `getConnectionFromDriver()`, 最后会调用到 `getConnectionFromDriverManager()`。你看一下这个方法, 里面就是我们熟悉的 `DriverManager.getConnection()`, 一层层调用, 最后还是落实到这里了。

所以我们看实际的数据库连接是什么时候创建的呢? 这个可以采用不同的策略, 可以在初始化 Bean 的时候创建, 也可以延后到实际使用的时候。MiniSpring 到现在这一步, 采取的是后面这个策略, 在应用程序 `dataSource.getConnection()` 的时候才实际生成数据库连接。

小结

我们这节课通过三个手段叠加，简化了数据库操作，重构了数据访问的程序结构。第一个手段是**模板化**，把通用代码写到一个 JdbcTemplate 模板里，把变化的部分交给具体的类来实现。第二个手段就是通过 **Callback 模式**，把具体类里实现的业务逻辑包装成一个回调函数，作为参数传给 JdbcTemplate 模板，这样就省去了要为每一个数据表单独增加一个具体实现类的工作。第三个手段就是结合 IoC 容器，**把 JdbcTemplate 声明成一个 Bean**，并利用 @Autowired 注解进行自动注入。

之后我们抽取出了数据源的概念，包装 connection，让应用程序和底下的数据库分隔开。

当然，程序走到这一步，还是有很多不足，主要的就是 JdbcTemplate 中还保留了很多固定的代码，比如 SQL 结果和业务对象的自动匹配问题，而且也没有考虑数据库连接池等等。这些都需要我们在后面的课程中一个个解决。

完整源代码参见 <https://github.com/YaleGuo/minis>

课后题

学完这节课，我也给你留一道思考题。我们现在只实现了 query，想一想如果想要实现 update 应该如何做呢？欢迎你在留言区与我交流讨论，也欢迎你把这节课分享给需要的朋友。我们下节课见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (3)



peter

2023-04-11 来自北京

请教老师几个问题：

Q1: JDBC driver复杂吗？代码规模一般多大？能否以加餐形式讲一下driver？

Q2: JDBC Template还有用吗？

现在一般的开发都是SSM或SSH，不会用JDBC Template。

Q3: JDBC template能支持多大并发？

用JDBC template的话，一个数据库实例，比如一个mysql实例，能支持多大的并发量？200？

作者回复: driver基础功能也不是很复杂, 主要是了解数据库系统提供的api, 现代数据库都提供了java api, 就更简单了。我那个时候只有c语言的api, 所以还要用JNI技术来做。基础功能c代码记得我就几千行。

jdbc template很少直接使用了, 一般用mybatis或者jpa, 但是也有系统是用的。我个人认为了解jdbc template很有好处。这个话题比较大, 见解不同, 我自己不是很支持orm技术。我的观点, 因为底下的数据库系统是关系模型, 是一种代数演算, 跟对象模型天然有隔, 一味进行orm, 我觉得是对面向对象的滥用。我在我的《认识编程》一书中讲到了这个观点。

数据库并发数, 跟jdbc template关系不大, 是数据库系统和缓存系统主要决定的, 实际上, 包装越少性能越好, jdbc template就是一层薄薄的包装。



1



C.

2023-04-27 来自江苏

前几天有点忙, 这次补齐了。<https://github.com/caozhenyuan/mini-spring.git>。请看jdbc1、2、3分支



马儿

2023-04-11 来自四川

实现dml语句如果只是简单的实现就像最初的那一版拼接sql就可以实现了, 但是这样的话需要每次更新都手动拼接sql比较麻烦。如果想要传入相应的对象就更新, 可以利用本节课的callback来实现将相应的对象字段转换为sql语句的过程。但是这节课的结果可能整体上离最终感知不到sql还比较远, 如果需要完全不感知sql应该是一个类专门负责根据类属性的注解来自动映射。

作者回复: 目标不同, jdbc template不是为了完全不感知sql。minispring没有讨论orm这个议题。

