作者博客：http://jinnianshilongnian.iteye.com/ 欢迎访问

第七章　对 JDBC 的支持

7.1　概述

7.1.1　JDBC 回顾

传统应用程序开发中，进行 JDBC 编程是相当痛苦的，如下所示：

```
//cn.javass.spring.chapter7. TraditionalJdbcTest
@Test
public void test() throws Exception {
    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
        conn = getConnection();                    //1.获取JDBC连接
                                                   //2.声明SQL
        String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";
        pstmt = conn.prepareStatement(sql);        //3.预编译SQL
        ResultSet rs = pstmt.executeQuery();       //4.执行SQL
        process(rs);                               //5.处理结果集
        closeResultSet(rs);                        //5.释放结果集
        closeStatement(pstmt);                     //6.释放Statement
        conn.commit();                             //8.提交事务
    } catch (Exception e) {
        //9.处理异常并回滚事务
        conn.rollback();
        throw e;
    } finally {
        //10.释放JDBC连接，防止JDBC连接不关闭造成的内存泄漏
        closeConnection(conn);
    }
}
```

以上代码片段具有冗长、重复、容易忘记某一步骤从而导致出错、显示控制事务、显示处理受检查异常等等。

有朋友可能重构出自己的一套 JDBC 模板，从而能简化日常开发，但自己开发的 JDBC 模板不够通用，而且对于每一套 JDBC 模板实现都差不多，从而导致开发人员必须掌握每一套模板。

Spring JDBC 提供了一套 JDBC 抽象框架，用于简化 JDBC 开发，而且如果各个公司都使用该抽象框架，开发人员首先减少了学习成本，直接上手开发，如图 7-1 所示。

传统JDBC	Spring JDBC
1.获取JDBC连接 2.声明SQL 3.预编译SQL 4.执行SQL 5.处理结果集 6.释放结果集 7.释放Statement 8.提交事务 9.处理异常并回滚事务 10.释放JDBC连接	~~1.获取JDBC连接~~ **2.声明SQL**（√） ~~3.预编译SQL~~ ~~4.执行SQL~~ **5.处理结果集**（√） ~~6.释放结果集~~ ~~7.释放Statement~~ ~~8.提交事务~~ ~~9.处理异常并回滚事务~~ ~~10.释放JDBC连接~~
缺点： 1.冗长、重复 2.显示事务控制 3.每个步骤不可获取 4.显示处理受检查异常	优点： 1.简单、简洁 2.Spring事务管理 3.只做需要做的 4.一致的非检查异常体系

图 7-1 Spring JDBC 与传统 JDBC 编程对比

7.1.2 Spring 对 JDBC 的支持

Spring 通过抽象 JDBC 访问并提供一致的 API 来简化 JDBC 编程的工作量。我们只需要**声明 SQL、调用合适的 Spring JDBC 框架 API、处理结果集**即可。事务由 Spring 管理，并将 JDBC 受查异常转换为 Spring 一致的非受查异常，从而简化开发。

Spring 主要提供 **JDBC 模板方式、关系数据库对象化方式和 SimpleJdbc** 方式三种方式来简化 JDBC 编程，这三种方式就是 Spring JDBC 的工作模式：

- **JDBC 模板方式：**Spring JDBC 框架提供以下几种模板类来简化 JDBC 编程，实现 GoF 模板设计模式，将可变部分和非可变部分分离，可变部分采用回调接口方式由用户来实现：如 JdbcTemplate、NamedParameterJdbcTemplate、SimpleJdbcTemplate。
- **关系数据库操作对象化方式：**Spring JDBC 框架提供了将关系数据库操作对象化的表示形式，从而使用户可以采用面向对象编程来完成对数据库的访问；

如 MappingSqlQuery、SqlUpdate、SqlCall、SqlFunction、StoredProcedure 等类。这些类的实现一旦建立即可重用并且是线程安全的。

- **SimpleJdbc 方式：**Spring JDBC 框架还提供了 **SimpleJdbc 方式**来简化 JDBC 编程，SimpleJdbcInsert 、 SimpleJdbcCall 用来简化数据库表插入、存储过程或函数访问。

Spring JDBC 还提供了一些强大的工具类，如 DataSourceUtils 来在必要的时候手工获取数据库连接等。

7.1.4 Spring 的 JDBC 架构

Spring JDBC 抽象框架由四部分组成：datasource、support、core、object。如图 7-2 所示。

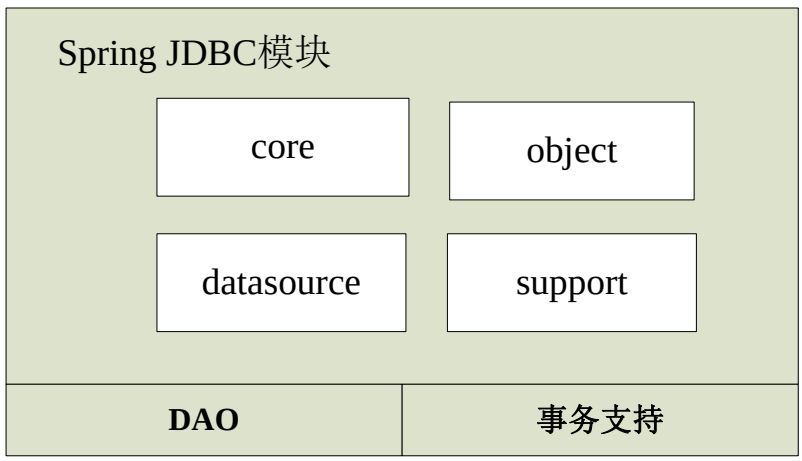

图 7-2 Spring JDBC 架构图

support 包：提供将 JDBC 异常转换为 DAO 非检查异常转换类、一些工具类如 JdbcUtils 等。

datasource 包：提供简化访问 JDBC 数据源（javax.sql.DataSource 实现）工具类，并提供了一些 DataSource 简单实现类从而能使从这些 DataSource 获取的连接能自动得到 Spring 管理事务支持。

core 包：提供 JDBC 模板类实现及可变部分的回调接口，还提供 SimpleJdbcInsert 等简单辅助类。

object 包：提供关系数据库的对象表示形式，如 MappingSqlQuery、SqlUpdate、SqlCall、SqlFunction、StoredProcedure 等类，该包是基于 core 包 JDBC 模板类实现。

7.2 JDBC 模板类

7.2.1 概述

Spring JDBC 抽象框架 core 包提供了 JDBC 模板类，其中 JdbcTemplate 是 core 包的核心类，所以其他模板类都是基于它封装完成的，JDBC 模板类是第一种工作模式。

JdbcTemplate 类通过模板设计模式帮助我们消除了冗长的代码，只做需要做的事情（即可变部分），并且帮我们做哪些固定部分，如连接的创建及关闭、。

JdbcTemplate 类对可变部分采用回调接口方式实现，如 ConnectionCallback 通过回调接口返回给用户一个连接，从而可以使用该连接做任何事情、StatementCallback 通过回调接口返回给用户一个 Statement，从而可以使用该 Statement 做任何事情等等，还有其他一些回调接口如图 7-3 所示。

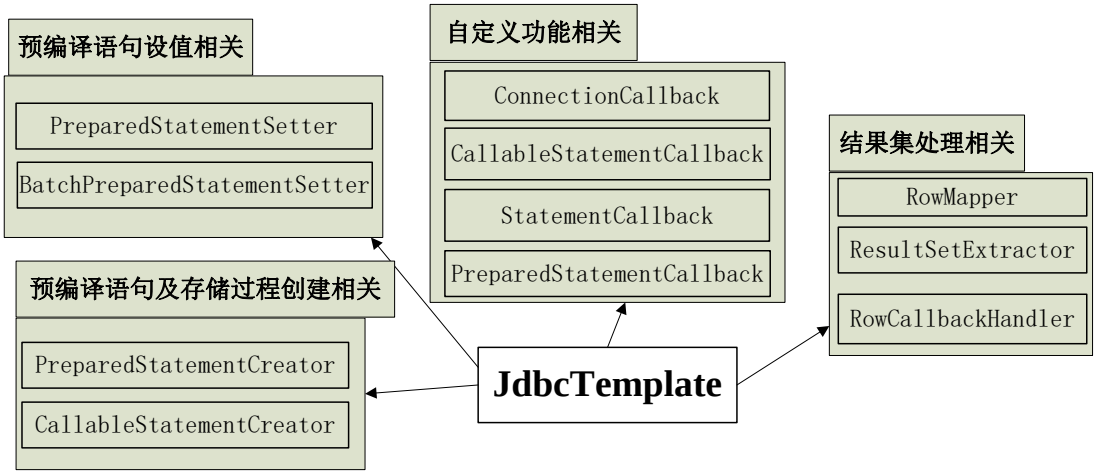

图 7-3 JdbcTemplate 支持的回调接口

Spring 除了提供 JdbcTemplate 核心类，还提供了基于 JdbcTemplate 实现的 NamedParameterJdbcTemplate 类用于支持命名参数绑定、SimpleJdbcTemplate 类用于支持 Java5+的可变参数及自动装箱拆箱等特性。

7.2.3 传统 JDBC 编程替代方案

前边我们已经使用过传统 JDBC 编程方式，接下来让我们看下 Spring JDBC 框架提供的更好的解决方案。

1）准备需要的 jar 包并添加到类路径中：

```
//JDBC 抽象框架模块
org.springframework.jdbc-3.0.5.RELEASE.jar
//Spring 事务管理及一致的 DAO 访问及非检查异常模块
org.springframework.transaction-3.0.5.RELEASE.jar
//hsqldb 驱动，hsqldb 是一个开源的 Java 实现数据库，请下载 hsqldb 2.0.0+版本
hsqldb.jar
```

2）传统 JDBC 编程替代方案：

在使用 JdbcTemplate 模板类时必须通过 DataSource 获取数据库连接，Spring JDBC 提供了 DriverManagerDataSource 实现，它通过包装“DriverManager.getConnection”获取数据库连接，具体 DataSource 相关请参考【7.5.1 控制数据库连接】。

```
package cn.javass.spring.chapter7;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
public class JdbcTemplateTest {
    private static JdbcTemplate jdbcTemplate;
    @BeforeClass
    public static void setUpClass() {
        String url = "jdbc:hsqldb:mem:test";
        String username = "sa";
        String password = "";
        DriverManagerDataSource dataSource =
            new DriverManagerDataSource(url, username, password);
        dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
    @Test
    public void test() {
        //1.声明SQL
        String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";
        jdbcTemplate.query(sql, new RowCallbackHandler() {
            @Override
            public void processRow(ResultSet rs) throws SQLException {
                //2.处理结果集
                String value = rs.getString("TABLE_NAME");
                System.out.println("Column TABLENAME:" + value);
            }
        });
    }
}
```

接下来让我们具体分析一下：

1） **jdbc:hsqldb:mem:test：**表示使用 hsqldb 内存数据库，数据库名为“test”。

2） **public static void setUpClass()：**使用 junit 的@BeforeClass 注解，表示在所以测试方法之前执行，且只执行一次。在此方法中定义了 DataSource 并使用 DataSource 对象创建了 JdbcTemplate 对象。JdbcTemplate 对象是线程安全的。

3） **JdbcTemplate 执行流程：**首先定义 SQL，其次调用 JdbcTemplate 方法执行 SQL，最后通过 RowCallbackHandler 回调处理 ResultSet 结果集。

Spring JDBC 解决方法相比传统 JDBC 编程方式是不是简单多了，是不是只有可变部分需要我们来做，其他的都由 Spring JDBC 框架来实现了。

接下来让我们深入 JdbcTemplate 及其扩展吧。

7.2.4 JdbcTemplate

首先让我们来看下如何使用 JdbcTemplate 来实现增删改查。

一、首先创建表结构：

```
//代码片段(cn.javass.spring.chapter7.JdbcTemplateTest)
@Before
public void setUp() {
   String createTableSql = "create memory table test" +
      "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
      "name varchar(100))";
   jdbcTemplate.update(createTableSql);
}
@After
public void tearDown() {
   String dropTableSql = "drop table test";
   jdbcTemplate.execute(dropTableSql);
```

1） org.junit 包下的**@Before 和@After** 分别表示在测试方法之前和之后执行的方法，对于每个测试方法都将执行一次；

2） **create memory table test** 表示创建 hsqldb 内存表，包含两个字段 id 和 name，其中 id 是具有自增功能的主键，如果有朋友对此不熟悉 hsqldb 可以换成熟悉的数据库。

二、定义测试骨架，该测试方法将用于实现增删改查测试：

```
@Test
public void testCURD() {
      insert();
      delete();
      update();
```

三、新增测试：

```
private void insert() {
    jdbcTemplate.update("insert into test(name) values('name1')");
    jdbcTemplate.update("insert into test(name) values('name2')");
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

四、删除测试：

```
private void delete() {
    jdbcTemplate.update("delete from test where name=?", new Object[]{"name2"});
    Assert.assertEquals(1, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

五、更新测试：

```
private void update() {
    jdbcTemplate.update("update test set name='name3' where name=?",
                        new Object[]{"name1"});
    Assert.assertEquals(1, jdbcTemplate.queryForInt("select count(*) from test
                                                    where name='name3'"));
}
```

六、查询测试：

```
private void select() {
    jdbcTemplate.query("select * from test", new RowCallbackHandler(){
        @Override
        public void processRow(ResultSet rs) throws SQLException {
            System.out.print("====id:" + rs.getInt("id"));
            System.out.println(",name:" + rs.getString("name"));
        }
    });
}
```

看完以上示例，大家是否觉得 JdbcTemplate 简化了我们很多劳动力呢？接下来让我们深入学习一下 JdbcTemplate 提供的方法。

JdbcTemplate 主要提供以下五类方法：

- **execute 方法：**可以用于执行任何 SQL 语句，一般用于执行 DDL 语句；
- **update 方法及 batchUpdate 方法：**update 方法用于执行新增、修改、删除等语句；batchUpdate 方法用于执行批处理相关语句；
- **query 方法及 queryForXXX 方法：**用于执行查询相关语句；
- **call 方法：**用于执行存储过程、函数相关语句。

JdbcTemplate 类支持的回调类：

- **预编译语句及存储过程创建回调：**用于根据 JdbcTemplate 提供的连接创建相应的语句；
  - PreparedStatementCreator：通过回调获取 JdbcTemplate 提供的 Connection，由用户使用该 Conncetion 创建相关的 PreparedStatement；
  - CallableStatementCreator：通过回调获取 JdbcTemplate 提供的 Connection，由用户使用该 Conncetion 创建相关的 CallableStatement；
- **预编译语句设值回调：**用于给预编译语句相应参数设值；
  - **PreparedStatementSetter：**通过回调获取 JdbcTemplate 提供的 PreparedStatement，由用户来对相应的预编译语句相应参数设值；
  - **BatchPreparedStatementSetter：**；类似于 PreparedStatementSetter，但用于批处理，需要指定批处理大小；
- **自定义功能回调：**提供给用户一个扩展点，用户可以在指定类型的扩展点执行任何数量需要的操作；
  - **ConnectionCallback：**通过回调获取 JdbcTemplate 提供的 Connection，用户可在该 Connection 执行任何数量的操作；
  - **StatementCallback：**通过回调获取 JdbcTemplate 提供的 Statement，用户可以在该 Statement 执行任何数量的操作；
  - **PreparedStatementCallback：**通过回调获取 JdbcTemplate 提供的 PreparedStatement，用户可以在该 PreparedStatement 执行任何数量的操作；
  - **CallableStatementCallback：**通过回调获取 JdbcTemplate 提供的 CallableStatement，用户可以在该 CallableStatement 执行任何数量的操作；
- **结果集处理回调：**通过回调处理 ResultSet 或将 ResultSet 转换为需要的形式；
  - **RowMapper：**用于将结果集每行数据转换为需要的类型，用户需实现方法 mapRow(ResultSet rs, int rowNum)来完成将每行数据转换为相应的类型。
  - **RowCallbackHandler：**用于处理 ResultSet 的每一行结果，用户需实现方法 processRow(ResultSet rs)来完成处理，在该回调方法中无需执行 rs.next()，该操作由 JdbcTemplate 来执行，用户只需按行获取数据然后处理即可。

- **ResultSetExtractor**：用于结果集数据提取，用户需实现方法 extractData(ResultSet rs)来处理结果集，用户必须处理整个结果集；

接下来让我们看下具体示例吧，在示例中不可能介绍到 JdbcTemplate 全部方法及回调类的使用方法，我们只介绍代表性的，其余的使用都是类似的；

1）预编译语句及存储过程创建回调、自定义功能回调使用：

```
@Test
public void testPpreparedStatement1() {
    int count = jdbcTemplate.execute(new PreparedStatementCreator() {
        @Override
        public PreparedStatement createPreparedStatement(Connection conn)
                throws SQLException {
            return conn.prepareStatement("select count(*) from test");
        }}, new PreparedStatementCallback<Integer>() {
        @Override
        public Integer doInPreparedStatement(PreparedStatement pstmt)
                throws SQLException, DataAccessException {
            pstmt.execute();
            ResultSet rs = pstmt.getResultSet();
            rs.next();
            return rs.getInt(1);
         }});
    Assert.assertEquals(0, count);
}
```

首先使用 PreparedStatementCreator 创建一个预编译语句，其次由 JdbcTemplate 通过 PreparedStatementCallback 回调传回，由用户决定如何执行该 PreparedStatement。此处我们使用的是 execute 方法。

2）预编译语句设值回调使用：

```
@Test
public void testPreparedStatement2() {
   String insertSql = "insert into test(name) values (?)";
   int count = jdbcTemplate.update(insertSql, new PreparedStatementSetter() {
        @Override
        public void setValues(PreparedStatement pstmt) throws SQLException {
            pstmt.setObject(1, "name4");
   }});
   Assert.assertEquals(1, count);
   String deleteSql = "delete from test where name=?";
   count = jdbcTemplate.update(deleteSql, new Object[] {"name4"});
   Assert.assertEquals(1, count);
```

通过 JdbcTemplate 的 int update(String sql, PreparedStatementSetter pss)执行预编译 sql，其中 sql 参数为“insert into test(name) values (?) ”，该 sql 有一个占位符需要在执行前设值，PreparedStatementSetter 实现就是为了设值，使用 setValues(PreparedStatement pstmt)回调方法设值相应的占位符位置的值。JdbcTemplate 也提供一种更简单的方式“update(String sql, Object... args)”来实现设值，所以只要当使用该种方式不满足需求时才应使用 PreparedStatementSetter。

3）结果集处理回调：

```
@Test
public void testResultSet1() {
   jdbcTemplate.update("insert into test(name) values('name5')");
   String listSql = "select * from test";
   List result = jdbcTemplate.query(listSql, new RowMapper<Map>() {
         @Override
         public Map mapRow(ResultSet rs, int rowNum) throws SQLException {
               Map row = new HashMap();
               row.put(rs.getInt("id"), rs.getString("name"));
               return row;
   }});
   Assert.assertEquals(1, result.size());
   jdbcTemplate.update("delete from test where name='name5'");
}
```

RowMapper 接口提供 mapRow(ResultSet rs, int rowNum)方法将结果集的每一行转换为一个 Map，当然可以转换为其他类，如表的对象画形式。

```
@Test
public void testResultSet2() {
   jdbcTemplate.update("insert into test(name) values('name5')");
   String listSql = "select * from test";
   final List result = new ArrayList();
   jdbcTemplate.query(listSql, new RowCallbackHandler() {
         @Override
         public void processRow(ResultSet rs) throws SQLException {
               Map row = new HashMap();
               row.put(rs.getInt("id"), rs.getString("name"));
               result.add(row);
   }});
   Assert.assertEquals(1, result.size());
```

RowCallbackHandler 接口也提供方法 processRow(ResultSet rs)，能将结果集的行转换为需要的形式。

```
@Test
public void testResultSet3() {
   jdbcTemplate.update("insert into test(name) values('name5')");
   String listSql = "select * from test";
   List result = jdbcTemplate.query(listSql, new ResultSetExtractor<List>() {
        @Override
        public List extractData(ResultSet rs)
             throws SQLException, DataAccessException {
             List result = new ArrayList();
             while(rs.next()) {
                  Map row = new HashMap();
                  row.put(rs.getInt("id"), rs.getString("name"));
                  result.add(row);
              }
              return result;
   }});
   Assert.assertEquals(0, result.size());
   jdbcTemplate.update("delete from test where name='name5'");
}
```

ResultSetExtractor 使用回调方法 extractData(ResultSet rs)提供给用户整个结果集，让用户决定如何处理该结果集。

当然 JdbcTemplate 提供更简单的 queryForXXX 方法，来简化开发：

```
//1.查询一行数据并返回 int 型结果
jdbcTemplate.queryForInt("select count(*) from test");
//2. 查询一行数据并将该行数据转换为 Map 返回
jdbcTemplate.queryForMap("select * from test where name='name5'");
//3.查询一行任何类型的数据，最后一个参数指定返回结果类型
jdbcTemplate.queryForObject("select count(*) from test", Integer.class);
//4.查询一批数据，默认将每行数据转换为 Map
jdbcTemplate.queryForList("select * from test");
//5.只查询一列数据列表，列类型是 String 类型，列名字是 name
jdbcTemplate.queryForList("
    select name from test where name=?", new Object[]{"name5"}, String.class);
```

3） 存储过程及函数回调：

首先修改 JdbcTemplateTest 的 setUp 方法，修改后如下所示：

```
@Before
public void setUp() {
    String createTableSql = "create memory table test" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "name varchar(100))";
    jdbcTemplate.update(createTableSql);

    String createHsqldbFunctionSql =
      "CREATE FUNCTION FUNCTION_TEST(str CHAR(100)) " +
      "returns INT begin atomic return length(str);end";
    jdbcTemplate.update(createHsqldbFunctionSql);
    String createHsqldbProcedureSql =
      "CREATE PROCEDURE PROCEDURE_TEST" +
      "(INOUT inOutName VARCHAR(100), OUT outId INT) " +
      "MODIFIES SQL DATA " +
      "BEGIN ATOMIC " +
      "    insert into test(name) values (inOutName); " +
      "    SET outId = IDENTITY(); " +
      "    SET inOutName = 'Hello,' + inOutName; " +
    "END";
      jdbcTemplate.execute(createHsqldbProcedureSql);
}
```

其中 CREATE FUNCTION FUNCTION_TEST 用于创建自定义函数，CREATE PROCEDURE PROCEDURE_TEST 用于创建存储过程，注意这些创建语句是数据库相关的，本示例中的语句只适用于 HSQLDB 数据库。

其次修改 JdbcTemplateTest 的 tearDown 方法，修改后如下所示：

```
@After
public void tearDown() {
    jdbcTemplate.execute("DROP FUNCTION FUNCTION_TEST");
    jdbcTemplate.execute("DROP PROCEDURE PROCEDURE_TEST");
    String dropTableSql = "drop table test";
    jdbcTemplate.execute(dropTableSql);
}
```

其中 drop 语句用于删除创建的存储过程、自定义函数及数据库表。

接下来看一下 hsqldb 如何调用自定义函数：

```
@Test
public void testCallableStatementCreator1() {
    final String callFunctionSql = "{call FUNCTION_TEST(?)}";
    List<SqlParameter> params = new ArrayList<SqlParameter>();
    params.add(new SqlParameter(Types.VARCHAR));
    params.add(new SqlReturnResultSet("result",
        new ResultSetExtractor<Integer>() {
            @Override
            public Integer extractData(ResultSet rs) throws SQLException,
                DataAccessException {
                while(rs.next()) {
                    return rs.getInt(1);
                }
                return 0;
        }));
    Map<String, Object> outValues = jdbcTemplate.call(
        new CallableStatementCreator() {
            @Override
            public CallableStatement createCallableStatement(Connection conn)
                throws SQLException {
                CallableStatement cstmt = conn.prepareCall(callFunctionSql);
                cstmt.setString(1, "test");
                return cstmt;
    }}, params);
    Assert.assertEquals(4, outValues.get("result"));
}
```

- **{call FUNCTION_TEST(?)}：**定义自定义函数的 sql 语句，注意 hsqldb {?= call …}和{call …}含义是一样的，而比如 mysql 中两种含义是不一样的；
- params：用于描述自定义函数占位符参数或命名参数类型；SqlParameter 用于描述 IN 类型参数、SqlOutParameter 用于描述 OUT 类型参数、SqlInOutParameter 用于

描述 INOUT 类型参数、SqlReturnResultSet 用于描述调用存储过程或自定义函数返回的 ResultSet 类型数据，其中 SqlReturnResultSet 需要提供结果集处理回调用于将结果集转换为相应的形式，hsqldb 自定义函数返回值是 ResultSet 类型。

- **CallableStatementCreator：**提供 Connection 对象用于创建 CallableStatement 对象
- outValues：调用 call 方法将返回类型为 Map<String, Object>对象；
- **outValues.get("result")：**获取结果，即通过 SqlReturnResultSet 对象转换过的数据；其中 SqlOutParameter、SqlInOutParameter、SqlReturnResultSet 指定的 name 用于从 call 执行后返回的 Map 中获取相应的结果，即 name 是 Map 的键。

注：因为 hsqldb {?= call …}和{call …}含义是一样的，因此调用自定义函数将返回一个包含结果的 ResultSet。

最后让我们示例下 mysql 如何调用自定义函数：

```
@Test
public void testCallableStatementCreator2() {
    JdbcTemplate mysqlJdbcTemplate = new JdbcTemplate(getMysqlDataSource);
    //2.创建自定义函数
    String createFunctionSql =
        "CREATE FUNCTION FUNCTION_TEST(str VARCHAR(100)) " +
         "returns INT return LENGTH(str)";
    String dropFunctionSql = "DROP FUNCTION IF EXISTS FUNCTION_TEST";
    mysqlJdbcTemplate.update(dropFunctionSql);
    mysqlJdbcTemplate.update(createFunctionSql);
    //3.准备sql,mysql支持{?= call …}
    final String callFunctionSql = "{?= call FUNCTION_TEST(?)}";
    //4.定义参数
    List<SqlParameter> params = new ArrayList<SqlParameter>();
    params.add(new SqlOutParameter("result", Types.INTEGER));
    params.add(new SqlParameter("str", Types.VARCHAR));
    Map<String, Object> outValues = mysqlJdbcTemplate.call(
      new CallableStatementCreator() {
        @Override
        public CallableStatement createCallableStatement(Connection conn)
          throws SQLException {
          CallableStatement cstmt = conn.prepareCall(callFunctionSql);
          cstmt.registerOutParameter(1, Types.INTEGER);
          cstmt.setString(2, "test");
        return cstmt;
    }}, params);
   Assert.assertEquals(4, outValues.get("result"));
}
public DataSource getMysqlDataSource() {
    String url = "jdbc:mysql://localhost:3306/test";
    DriverManagerDataSource dataSource =
        new DriverManagerDataSource(url, "root", "");
```

- **getMysqlDataSource：**首先启动 mysql（本书使用 5.4.3 版本），其次登录 mysql 创建 test 数据库（“create database test;”），在进行测试前，请先下载并添加 mysql-connector-java-5.1.10.jar 到 classpath；
- **{?= call FUNCTION_TEST(?)}：**可以使用{?= call …}形式调用自定义函数；
- **params：**无需使用 SqlReturnResultSet 提取结果集数据，而是使用 SqlOutParameter 来描述自定义函数返回值；
- **CallableStatementCreator：**同上个例子含义一样；
- **cstmt.registerOutParameter(1, Types.INTEGER)：**将 OUT 类型参数注册为 JDBC 类型 Types.INTEGER，此处即返回值类型为 Types.INTEGER。
- **outValues.get("result")：**获取结果，直接返回 Integer 类型，比 hsqldb 简单多了吧。

最后看一下如何如何调用存储过程：

```
@Test
public void testCallableStatementCreator3() {
    final String callProcedureSql = "{call PROCEDURE_TEST(?, ?)}";
    List<SqlParameter> params = new ArrayList<SqlParameter>();
    params.add(new SqlInOutParameter("inOutName", Types.VARCHAR));
    params.add(new SqlOutParameter("outId", Types.INTEGER));
    Map<String, Object> outValues = jdbcTemplate.call(
      new CallableStatementCreator() {
        @Override
        public CallableStatement createCallableStatement(Connection conn)
          throws SQLException {
          CallableStatement cstmt = conn.prepareCall(callProcedureSql);
          cstmt.registerOutParameter(1, Types.VARCHAR);
          cstmt.registerOutParameter(2, Types.INTEGER);
          cstmt.setString(1, "test");
          return cstmt;
    }}, params);
    Assert.assertEquals("Hello,test", outValues.get("inOutName"));
    Assert.assertEquals(0, outValues.get("outId"));
}
```

- **{call PROCEDURE_TEST(?, ?)}**：定义存储过程 sql；
- **params**：定义存储过程参数；SqlInOutParameter 描述 INOUT 类型参数、SqlOutParameter 描述 OUT 类型参数；
- CallableStatementCreator：用于创建 CallableStatement，并设值及注册 OUT 参数类型；
- outValues：通过 SqlInOutParameter 及 SqlOutParameter 参数定义的 name 来获取存储过程结果。

JdbcTemplate 类还提供了很多便利方法，在此就不一一介绍了，但这些方法是由规律可循的，第一种就是提供回调接口让用户决定做什么，第二种可以认为是便利方法（如 queryForXXX），用于那些比较简单的操作。

7.2.4 NamedParameterJdbcTemplate

NamedParameterJdbcTemplate 类是基于 JdbcTemplate 类，并对它进行了封装从而支持命名参数特性。

NamedParameterJdbcTemplate 主要提供以下三类方法：execute 方法、query 及 queryForXXX 方法、update 及 batchUpdate 方法。

首先让我们看个例子吧：

```
@Test
public void testNamedParameterJdbcTemplate1() {
    NamedParameterJdbcTemplate namedParameterJdbcTemplate = null;
    //namedParameterJdbcTemplate =
    //      new NamedParameterJdbcTemplate(dataSource);
    namedParameterJdbcTemplate =
          new NamedParameterJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(name) values(:name)";
    String selectSql = "select * from test where name=:name";
    String deleteSql = "delete from test where name=:name";
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("name", "name5");
    namedParameterJdbcTemplate.update(insertSql, paramMap);
    final List<Integer> result = new ArrayList<Integer>();
    namedParameterJdbcTemplate.query(selectSql, paramMap,
      new RowCallbackHandler() {
        @Override
            public void processRow(ResultSet rs) throws SQLException {
                result.add(rs.getInt("id"));
            }
    });
    Assert.assertEquals(1, result.size());
```

接下来让我们分析一下代码吧：

1）NamedParameterJdbcTemplate 初始化：可以使用 DataSource 或 JdbcTemplate 对象作为构造器参数初始化；

2）insert into test(name) values(:name)：其中“:name”就是命名参数；

3） update(insertSql, paramMap)：其中 paramMap 是一个 Map 类型，包含键为“name”，值为“name5”的键值对，也就是为命名参数设值的数据；

4）query(selectSql, paramMap, new RowCallbackHandler()……)：类似于 JdbcTemplate中介绍的，唯一不同是需要传入paramMap来为命名参数设值；

5）update(deleteSql, paramSource)：类似于“update(insertSql, paramMap)”，但使用 SqlParameterSource 参数来为命名参数设值，此处使用 MapSqlParameterSource 实现，其就是简单封装 java.util.Map。

NamedParameterJdbcTemplate 类为命名参数设值有两种方式：java.util.Map 和 SqlParameterSource：

1）java.util.Map：使用 Map 键数据来对于命名参数，而 Map 值数据用于设值；

2）SqlParameterSource：可以使用 SqlParameterSource 实现作为来实现为命名参数设值，默认有 MapSqlParameterSource 和 BeanPropertySqlParameterSource 实现；MapSqlParameterSource 实现非常简单，只是封装了 java.util.Map；而 BeanPropertySqlParameterSource 封装了一个 JavaBean 对象，通过 JavaBean 对象属性来决定命名参数的值。

```
package cn.javass.spring.chapter7;
public class UserModel {
    private int id;
    private String myName;
    //省略getter和setter
}
```

```
@Test
public void testNamedParameterJdbcTemplate2() {
    NamedParameterJdbcTemplate namedParameterJdbcTemplate = null;
    namedParameterJdbcTemplate =
        new NamedParameterJdbcTemplate(jdbcTemplate);
    UserModel model = new UserModel();
    model.setMyName("name5");
    String insertSql = "insert into test(name) values(:myName)";
    SqlParameterSource paramSource = new
        BeanPropertySqlParameterSource(model);
    namedParameterJdbcTemplate.update(insertSql, paramSource);
}
```

可以看出 BeanPropertySqlParameterSource 使用能减少很多工作量，但命名参数必须和 JavaBean 属性名称相对应才可以。

7.2.5 SimpleJdbcTemplate

SimpleJdbcTemplate 类也是基于 JdbcTemplate 类，但利用 Java5+的可变参数列表和自动装箱和拆箱从而获取更简洁的代码。

SimpleJdbcTemplate 主要提供两类方法：query 及 queryForXXX 方法、update 及 batchUpdate 方法。

首先让我们看个例子吧：

```
//定义 UserModel 的 RowMapper
package cn.javass.spring.chapter7;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
public class UserRowMapper implements RowMapper<UserModel> {
    @Override
    public UserModel mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        UserModel model = new UserModel();
        model.setId(rs.getInt("id"));
        model.setMyName(rs.getString("name"));
        return model;
    }
}
```

```
@Test
public void testSimpleJdbcTemplate() {
    //还支持DataSource和NamedParameterJdbcTemplate作为构造器参数
    SimpleJdbcTemplate simpleJdbcTemplate =
        new SimpleJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(id, name) values(?, ?)";
    simpleJdbcTemplate.update(insertSql, 10, "name5");
    String selectSql = "select * from test where id=? and name=?";
    List<Map<String, Object>> result =
        simpleJdbcTemplate.queryForList(selectSql, 10, "name5");
    Assert.assertEquals(1, result.size());
    RowMapper<UserModel> mapper = new UserRowMapper();
```

1）SimpleJdbcTemplate 初始化：可以使用 DataSource、JdbcTemplate 或 NamedParameterJdbcTemplate 对象作为构造器参数初始化；

2）update(insertSql, 10, "name5")：采用Java5+可变参数列表从而代替new Object[]{10, "name5"}方式；

3）query(selectSql, mapper, 10, "name5")：使用Java5+可变参数列表及RowMapper回调并利用泛型特性来指定返回值类型（List<UserModel>）。

SimpleJdbcTemplate类还支持命名参数特性，如queryForList(String sql, SqlParameterSource args)和queryForList(String sql, Map<String, ?> args) ，类似于NamedParameterJdbcTemplate中使用，在此就不介绍了。

注：SimpleJdbcTemplate 还提供类似于 ParameterizedRowMapper 用于泛型特性的支持，ParameterizedRowMapper 是 RowMapper 的子类，但从 Spring 3.0 由于允许环境需要 Java5+，因此不再需要 ParameterizedRowMapper ，而可以直接使用 RowMapper；

~~**query(String sql, ParameterizedRowMapper<T> rm, SqlParameterSource args)**~~

query(String sql, RowMapper<T> rm, Object... args) //直接使用该语句

SimpleJdbcTemplate 还提供如下方法用于获取 JdbcTemplate 和 NamedParameterJdbcTemplate：

1）获取 JdbcTemplate 对象方法：JdbcOperations 是 JdbcTemplate 的接口

JdbcOperations getJdbcOperations()

2）获取 NamedParameterJdbcTemplate 对象方法：NamedParameterJdbcOperations 是 NamedParameterJdbcTemplate 的接口

NamedParameterJdbcOperations getNamedParameterJdbcOperations()

7.3 关系数据库操作对象化

7.3.1 概述

所谓**关系数据库对象化其实就是用面向对象方式表示关系数据库操作，从而可以复用。**

Spring JDBC 框架将数据库操作封装为一个 RdbmsOperation，该对象是线程安全的、可复用的对象，是所有数据库对象的父类。而 SqlOperation 继承了 RdbmsOperation，代表了数据库 SQL 操作，如 select、update、call 等，如图 7-4 所示。

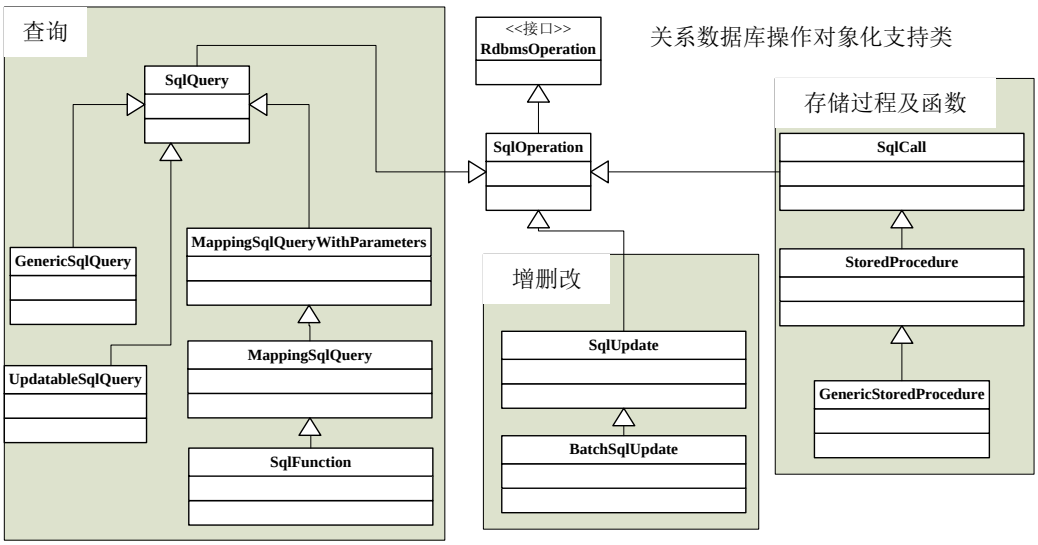

图 7-4 关系数据库操作对象化支持类

数据库操作对象化只要有以下几种类型，所以类型是线程安全及可复用的：

- **查询：**将数据库操作 select 封装为对象，查询操作的基类是 SqlQuery，所有查询都可以使用该类表示，Spring JDBC 还提供了一些更容易使用的 MappingSqlQueryWithParameters 和 MappingSqlQuery 用于将结果集映射为 Java 对象，查询对象类还提供了两个扩展 UpdatableSqlQuery 和 SqlFunction；
- **更新：**即增删改操作，将数据库操作 insert 、update、delete 封装为对象，增删改基类是 SqlUpdate，当然还提供了 BatchSqlUpdate 用于批处理；
- **存储过程及函数：**将存储过程及函数调用封装为对象，基类是 SqlCall 类，提供了 StoredProcedure 实现。

7.3.2 查询

1）SqlQuery：需要覆盖如下方法来定义一个 RowMapper，其中 parameters 参数表示命名参数或占位符参数值列表，而 context 是由用户传入的上下文数据。

```
RowMapper<T> newRowMapper(Object[] parameters, Map context)
```

SqlQuery 提供两类方法：

- execute 及 executeByNamedParam 方法：用于查询多行数据，其中 executeByNamedParam 用于支持命名参数绑定参数；
- findObject 及 findObjectByNamedParam 方法：用于查询单行数据，其中 findObjectByNamedParam 用于支持命名参数绑定。

演示一下 SqlQuery 如何使用：

```
@Test
public void testSqlQuery() {
    SqlQuery query = new UserModelSqlQuery(jdbcTemplate);
    List<UserModel> result = query.execute("name5");
    Assert.assertEquals(0, result.size());
}
```

从测试代码可以 SqlQuery 使用非常简单，创建 SqlQuery 实现对象，然后调用相应的方法即可，接下来看一下 SqlQuery 实现：

```
package cn.javass.spring.chapter7;
//省略 import
public class UserModelSqlQuery extends SqlQuery<UserModel> {
    public UserModelSqlQuery(JdbcTemplate jdbcTemplate) {
        //super.setDataSource(jdbcTemplate.getDataSource());
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("select * from test where name=?");
        super.declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }
    @Override
    protected RowMapper<UserModel> newRowMapper(Object[] parameters,
                                                Map context) {
        return new UserRowMapper();
    }
}
```

从测试代码可以看出，具体步骤如下：

一、setJdbcTemplate/ setDataSource：首先设置数据源或 JdbcTemplate；

二、setSql("select * from test where name=?")：定义 sql 语句，所以定义的 sql 语句都将被编译为 PreparedStatement；

三、declareParameter(new SqlParameter(Types.VARCHAR))：对 PreparedStatement 参数描述，使用 SqlParameter 来描述参数类型，支持命名参数、占位符描述；对于命名参数可以使用如 new SqlParameter("name", Types.VARCHAR)描述；注意占位符参数描述必须按占位符参数列表的顺序进行描述；

四、编译：可选，当执行相应查询方法时会自动编译，用于将 sql 编译为 PreparedStatement，对于编译的 SqlQuery 不能再对参数进行描述了。

五、以上步骤是不可变的，必须按顺序执行。

2）MappingSqlQuery：用于简化 SqlQuery 中 RowMapper 创建，可以直接在实现 mapRow(ResultSet rs, int rowNum)来将行数据映射为需要的形式；

MappingSqlQuery 所有查询方法完全继承于 SqlQuery。

演示一下 MappingSqlQuery 如何使用：

```
@Test
public void testMappingSqlQuery() {
    jdbcTemplate.update("insert into test(name) values('name5')");
    SqlQuery<UserModel> query = new
    UserModelMappingSqlQuery(jdbcTemplate);
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("name", "name5");
    UserModel result = query.findObjectByNamedParam(paramMap);
    Assert.assertNotNull(result);
}
```

MappingSqlQuery 使用和 SqlQuery 完全一样，创建 MappingSqlQuery 实现对象，然后调用相应的方法即可，接下来看一下 MappingSqlQuery 实现，findObjectByNamedParam 方法用于执行命名参数查询：

```
package cn.javass.spring.chapter7;
//省略 import
public class UserModelMappingSqlQuery extends MappingSqlQuery<UserModel> {
  public UserModelMappingSqlQuery(JdbcTemplate jdbcTemplate) {
    super.setDataSource(jdbcTemplate.getDataSource());
    super.setSql("select * from test where name=:name");
    super.declareParameter(new SqlParameter("name", Types.VARCHAR));
    compile();
  }
  @Override
  protected UserModel mapRow(ResultSet rs, int rowNum) throws SQLException {
    UserModel model = new UserModel();
    model.setId(rs.getInt("id"));
    model.setMyName(rs.getString("name"));
    return model;
  }
}
```

和 SqlQuery 唯一不同的是使用 mapRow 来讲每行数据转换为需要的形式，其他地方完

全一样。

4） **UpdatableSqlQuery：**提供可更新结果集查询支持，子类实现 updateRow(ResultSet rs, int rowNum, Map context)对结果集进行更新。

5） **GenericSqlQuery：**提供 setRowMapperClass(Class rowMapperClass)方法用于指定 RowMapper 实现，在此就不演示了。具体请参考 testGenericSqlQuery()方法。

6） **SqlFunction：**SQL“函数”包装器，用于支持那些返回单行结果集的查询。该类主要用于返回单行单列结果集。

```
@Test
public void testSqlFunction() {
    jdbcTemplate.update("insert into test(name) values('name5')");
    String countSql = "select count(*) from test";
    SqlFunction<Integer> sqlFunction1 =
        new SqlFunction<Integer>(jdbcTemplate.getDataSource(), countSql);
    Assert.assertEquals(1, sqlFunction1.run());
    String selectSql = "select name from test where name=?";
    SqlFunction<String> sqlFunction2 =
        new SqlFunction<String>(jdbcTemplate.getDataSource(), selectSql);
    sqlFunction2.declareParameter(new SqlParameter(Types.VARCHAR));
    String name = (String) sqlFunction2.runGeneric(new Object[] {"name5"});
    Assert.assertEquals("name5", name);
}
```

如代码所示，SqlFunction 初始化时需要 DataSource 和相应的 sql 语句，如果有参数需要使用 declareParameter 对参数类型进行描述；run 方法默认返回 int 型，当然也可以使用 runGeneric 返回其他类型，如 String 等。

7.3.3 更新

SqlUpdate 类用于支持数据库更新操作，即增删改（insert、delete、update）操作，该方法类似于 SqlQuery，只是职责不一样。

SqlUpdate 提供了 update 及 updateByNamedParam 方法用于数据库更新操作，其中 updateByNamedParam 用于命名参数类型更新。

演示一下 SqlUpdate 如何使用：

```
package cn.javass.spring.chapter7;
//省略import
public class InsertUserModel extends SqlUpdate {
    public InsertUserModel(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("insert into test(name) values(?)");
        super.declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }
}
```

```
@Test
public void testSqlUpdate() {
    SqlUpdate insert = new InsertUserModel(jdbcTemplate);
    insert.update("name5");

    String updateSql = "update test set name=? where name=?";
    SqlUpdate update = new SqlUpdate(jdbcTemplate.getDataSource(),
                    updateSql, new int[]{Types.VARCHAR, Types.VARCHAR});
    update.update("name6", "name5");
    String deleteSql = "delete from test where name=:name";

    SqlUpdate delete = new SqlUpdate(jdbcTemplate.getDataSource(),
                        deleteSql, new int[]{Types.VARCHAR});
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("name", "name5");
    delete.updateByNamedParam(paramMap);
}
```

InsertUserModel 类实现类似于 SqlQuery 实现，用于执行数据库插入操作，SqlUpdate 还提供一种更简洁的构造器 SqlUpdate(DataSource ds, String sql, int[] types)，其中 types 用于指定占位符或命名参数类型；SqlUpdate 还支持命名参数，使用 updateByNamedParam 方法来进行命名参数操作。

7.3.4 存储过程及函数

StoredProcedure 用于支持存储过程及函数，该类的使用同样类似于 SqlQuery。

StoredProcedure 提供 execute 方法用于执行存储过程及函数。

一、StoredProcedure 如何调用自定义函数：

```
@Test
public void testStoredProcedure1() {
    StoredProcedure lengthFunction = new HsqldbLengthFunction(jdbcTemplate);
    Map<String,Object> outValues = lengthFunction.execute("test");
    Assert.assertEquals(4, outValues.get("result"));
}
```

StoredProcedure 使用非常简单，定义 StoredProcedure 实现 HsqldbLengthFunction，并调用 execute 方法执行即可，接下来看一下 HsqldbLengthFunction 实现：

```
package cn.javass.spring.chapter7;
//省略import
public class HsqldbLengthFunction extends StoredProcedure {
    public HsqldbLengthFunction(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("FUNCTION_TEST");
        super.declareParameter(
            new SqlReturnResultSet("result", new ResultSetExtractor<Integer>() {
            @Override
            public Integer extractData(ResultSet rs)
                throws SQLException, DataAccessException {
                while(rs.next()) {
                    return rs.getInt(1);
                }
                return 0;
            }
        }));
        super.declareParameter(new SqlParameter("str", Types.VARCHAR));
        compile();
    }
}
```

StoredProcedure 自定义函数使用类似于 SqlQuery，首先设置数据源或 JdbcTemplate 对象，其次定义自定义函数，然后使用 declareParameter 进行参数描述，最后调用 compile（可选）编译自定义函数。

接下来看一下 mysql 自定义函数如何使用：

```
@Test
public void testStoredProcedure2() {
    JdbcTemplate mysqlJdbcTemplate = new JdbcTemplate(getMysqlDataSource());
    String createFunctionSql =
        "CREATE FUNCTION FUNCTION_TEST(str VARCHAR(100)) " +
        "returns INT return LENGTH(str)";
    String dropFunctionSql = "DROP FUNCTION IF EXISTS FUNCTION_TEST";
    mysqlJdbcTemplate.update(dropFunctionSql);
    mysqlJdbcTemplate.update(createFunctionSql);
    StoredProcedure lengthFunction =
        new MysqlLengthFunction(mysqlJdbcTemplate);
    Map<String,Object> outValues = lengthFunction.execute("test");
    Assert.assertEquals(4, outValues.get("result"));
}
```

MysqlLengthFunction 自定义函数使用与 HsqldbLengthFunction 使用完全一样，只是内部实现稍有差别：

```
package cn.javass.spring.chapter7;
//省略 import
public class MysqlLengthFunction extends StoredProcedure {
    public MysqlLengthFunction(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("FUNCTION_TEST");
        super.setFunction(true);
        super.declareParameter(new SqlOutParameter("result", Types.INTEGER));
        super.declareParameter(new SqlParameter("str", Types.VARCHAR));
        compile();
    }
}
```

MysqlLengthFunction 与 HsqldbLengthFunction 实现不同的地方有两点：

- **setFunction(true)**：表示是自定义函数调用，即编译后的 sql 为{?= call …}形式；如果使用 hsqldb 不能设置为 true，因为在 hsqldb 中{?= call …}和{call …}含义一样；
- **declareParameter(new SqlOutParameter("result", Types.INTEGER))**：将自定义函数返回值类型直接描述为 Types.INTEGER；SqlOutParameter 必须指定 name，而不用使用 SqlReturnResultSet 首先获取结果集，然后再从结果集获取返回值，这

是 mysql 与 hsqldb 的区别；

一、StoredProcedure 如何调用存储过程：

```
@Test
public void testStoredProcedure3() {
    StoredProcedure procedure = new HsqldbTestProcedure(jdbcTemplate);
    Map<String,Object> outValues = procedure.execute("test");
    Assert.assertEquals(0, outValues.get("outId"));
    Assert.assertEquals("Hello,test", outValues.get("inOutName"));
}
```

StoredProcedure 存储过程实现 HsqldbTestProcedure 调用与 HsqldbLengthFunction 调用完全一样，不同的是在实现时，参数描述稍有不同：

```
package cn.javass.spring.chapter7;
//省略 import
public class HsqldbTestProcedure extends StoredProcedure {
    public HsqldbTestProcedure(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("PROCEDURE_TEST");
        super.declareParameter(
                new SqlInOutParameter("inOutName", Types.VARCHAR));
        super.declareParameter(new SqlOutParameter("outId", Types.INTEGER));
        compile();
    }
}
```

- **declareParameter**：使用 SqlInOutParameter 描述 INOUT 类型参数，使用 SqlOutParameter 描述 OUT 类型参数，必须按顺序定义，不能颠倒。

7.4 Spring 提供的其它帮助

7.4.1 SimpleJdbc 方式

Spring JDBC 抽象框架提供 SimpleJdbcInsert 和 SimpleJdbcCall 类，这两个类通过利用 JDBC 驱动提供的数据库元数据来简化 JDBC 操作。

- SimpleJdbcInsert：用于插入数据，根据数据库元数据进行插入数据，本类用于简化插入操作，提供三种类型方法：execute 方法用于普通插入、executeAndReturnKey 及 executeAndReturnKeyHolder 方法用于插入时获取主键值、executeBatch 方法用于批

处理。

```
@Test
public void testSimpleJdbcInsert() {
    SimpleJdbcInsert insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    Map<String, Object> args = new HashMap<String, Object>();
    args.put("name", "name5");
    insert.compile();
    //1.普通插入
    insert.execute(args);
    Assert.assertEquals(1, jdbcTemplate.queryForInt("select count(*) from test"));
    //2.插入时获取主键值
    insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    insert.setGeneratedKeyName("id");
    Number id = insert.executeAndReturnKey(args);
    Assert.assertEquals(1, id);
    //3.批处理
    insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    insert.setGeneratedKeyName("id");
    int[] updateCount = insert.executeBatch(new Map[] {args, args, args});
    Assert.assertEquals(1, updateCount[0]);
    Assert.assertEquals(5, jdbcTemplate.queryForInt("select count(*) from test"));
```

- **new SimpleJdbcInsert(jdbcTemplate)**：首次通过 DataSource 对象或 JdbcTemplate 对象初始化 SimpleJdbcInsert；
- **insert.withTableName("test")**：用于设置数据库表名；
- **args**：用于指定插入时列名及值，如本例中只有 name 列名，即编译后的 sql 类似于“insert into test(name) values(?)”；
- **insert.compile()**：可选的编译步骤，在调用执行方法时自动编译，编译后不能再对 insert 对象修改；
- **执行**：execute 方法用于执行普通插入；executeAndReturnKey 用于执行并获取自动生成主键（注意是 Number 类型），必须首先通过 setGeneratedKeyName 设置主键然后才能获取，如果想获取复合主键请使用 setGeneratedKeyNames 描述主键然后通过 executeReturningKeyHolder 获取复合主键 KeyHolder 对象；executeBatch 用于批处理；

- SimpleJdbcCall：用于调用存储过程及自定义函数，本类用于简化存储过程及自定

义函数调用。

```
@Test
public void testSimpleJdbcCall1() {
    //此处用mysql,因为hsqldb调用自定义函数和存储过程一样
    SimpleJdbcCall call = new SimpleJdbcCall(getMysqlDataSource());
    call.withFunctionName("FUNCTION_TEST");
    call.declareParameters(new SqlOutParameter("result", Types.INTEGER));
    call.declareParameters(new SqlParameter("str", Types.VARCHAR));
    Map<String, Object> outVlaues = call.execute("test");
    Assert.assertEquals(4, outVlaues.get("result"));
}
```

- **new SimpleJdbcCall(getMysqlDataSource())**：通过 DataSource 对象或 JdbcTemplate 对象初始化 SimpleJdbcCall；
- **withFunctionName("FUNCTION_TEST")**：定义自定义函数名；自定义函数 sql 语句将被编译为类似于{?= call …}形式；
- **declareParameters**：描述参数类型，使用方式与 StoredProcedure 对象一样；
- **执行**：调用 execute 方法执行自定义函数；

```
@Test
public void testSimpleJdbcCall2() {
    //调用hsqldb自定义函数得使用如下方式
    SimpleJdbcCall call = new SimpleJdbcCall(jdbcTemplate);
    call.withProcedureName("FUNCTION_TEST");
    call.declareParameters(new SqlReturnResultSet("result",
      new ResultSetExtractor<Integer>() {
        @Override
        public Integer extractData(ResultSet rs)
          throws SQLException, DataAccessException {
          while(rs.next()) {
            return rs.getInt(1);
          }
          return 0;
    }}));
    call.declareParameters(new SqlParameter("str", Types.VARCHAR));
    Map<String, Object> outVlaues = call.execute("test");
    Assert.assertEquals(4, outVlaues.get("result"));
}
```

调用 hsqldb 数据库自定义函数与调用 mysql 自定义函数完全不同，详见

StoredProcedure 中的解释。

```
@Test
public void testSimpleJdbcCall3() {
   SimpleJdbcCall call = new SimpleJdbcCall(jdbcTemplate);
   call.withProcedureName("PROCEDURE_TEST");
   call.declareParameters(new SqlInOutParameter("inOutName", Types.VARCHAR));
   call.declareParameters(new SqlOutParameter("outId", Types.INTEGER));
   SqlParameterSource params =
      new MapSqlParameterSource().addValue("inOutName", "test");
   Map<String, Object> outVlaues = call.execute(params);
   Assert.assertEquals("Hello,test", outVlaues.get("inOutName"));
   Assert.assertEquals(0, outVlaues.get("outId"));
}
```

与自定义函数调用不同的是使用 withProcedureName 来指定存储过程名字；其他参数描述等完全一样。

7.4.1 控制数据库连接

Spring JDBC 通过 DataSource 控制数据库连接，即通过 DataSource 实现获取数据库连接。

Spring JDBC 提供了一下 DataSource 实现：

- **DriverManagerDataSource**：简单封装了 DriverManager 获取数据库连接；通过 DriverManager 的 getConnection 方法获取数据库连接；
- **SingleConnectionDataSource**：内部封装了一个连接，该连接使用后不会关闭，且不能在多线程环境中使用，一般用于测试；
- **LazyConnectionDataSourceProxy**：包装一个 DataSource，用于延迟获取数据库连接，只有在真正创建 Statement 等时才获取连接，因此再说实际项目中最后使用该代理包装原始 DataSource 从而使得只有在真正需要连接时才去获取。

第三方提供的 DataSource 实现主要有 C3P0、Proxool、DBCP 等，这些实现都具有数据库连接池能力。

DataSourceUtils：Spring JDBC 抽象框架内部都是通过它的 getConnection(DataSource dataSource) 方法获取数据库连接，releaseConnection(Connection con, DataSource dataSource) 用于释放数据库连接，DataSourceUtils 用于支持 Spring 管理事务，只有使用 DataSourceUtils 获取的连接才具有 Spring 管理事务。

7.4.3 获取自动生成的主键

有许多数据库提供自动生成主键的能力，因此我们可能需要获取这些自动生成的主键，JDBC 3.0 标准支持获取自动生成的主键，且必须数据库支持自动生成键获取。

1）JdbcTemplate 获取自动生成主键方式：

```
@Test
public void testFetchKey1() throws SQLException {
    final String insertSql = "insert into test(name) values('name5')";
    KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
    jdbcTemplate.update(new PreparedStatementCreator() {
        @Override
        public PreparedStatement createPreparedStatement(Connection conn)
            throws SQLException {
            return conn.prepareStatement(insertSql, new String[]{"ID"});
      }}, generatedKeyHolder);
    Assert.assertEquals(0, generatedKeyHolder.getKey());
}
```

使用 JdbcTemplate 的 update(final PreparedStatementCreator psc, final KeyHolder generatedKeyHolder)方法执行需要返回自动生成主键的插入语句，其中 psc 用于创建 PreparedStatement 并指定自动生成键，如“prepareStatement(insertSql, new String[]{"ID"})”；generatedKeyHolder 是 KeyHolder 类型，用于获取自动生成的主键或复合主键；如使用 getKey 方法获取自动生成的主键。

2）SqlUpdate 获取自动生成主键方式：

```
@Test
public void testFetchKey2() {
    final String insertSql = "insert into test(name) values('name5')";
    KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
    SqlUpdate update = new SqlUpdate();
    update.setJdbcTemplate(jdbcTemplate);
    update.setReturnGeneratedKeys(true);
    //update.setGeneratedKeysColumnNames(new String[]{"ID"});
    update.setSql(insertSql);
    update.update(null, generatedKeyHolder);
    Assert.assertEquals(0, generatedKeyHolder.getKey());
}
```

SqlUpdate 获取自动生成主键方式和 JdbcTemplate 完全一样，可以使用 setReturnGeneratedKeys（true）表示要获取自动生成键；也可以使用

setGeneratedKeysColumnNames 指定自动生成键列名。

3）SimpleJdbcInsert：前边示例已介绍，此处就不演示了。

7.4.4 JDBC 批量操作

JDBC 批处理用于减少与数据库交互的次数来提升性能，Spring JDBC 抽象框架通过封装批处理操作来简化批处理操作

1）JdbcTemplate 批处理：支持普通的批处理及占位符批处理；

```
@Test
public void testBatchUpdate1() {
    String insertSql = "insert into test(name) values('name5')";
    String[] batchSql = new String[] {insertSql, insertSql};
    jdbcTemplate.batchUpdate(batchSql);
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

直接调用 batchUpdate 方法执行需要批处理的语句即可。

```
@Test
public void testBatchUpdate2() {
    String insertSql = "insert into test(name) values(?)";
    final String[] batchValues = new String[] {"name5", "name6"};
    jdbcTemplate.batchUpdate(insertSql, new BatchPreparedStatementSetter() {
        @Override
        public void setValues(PreparedStatement ps, int i) throws SQLException {
            ps.setString(1, batchValues[i]);
        }
        @Override
        public int getBatchSize() {
            return batchValues.length;
        }
    });
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

JdbcTemplate 还可以通过 batchUpdate(String sql, final BatchPreparedStatementSetter pss) 方法进行批处理，该方式使用预编译语句，然后通过 BatchPreparedStatementSetter 实现进行设值（setValues）及指定批处理大小（getBatchSize）。

2）NamedParameterJdbcTemplate 批处理：支持命名参数批处理；

```
@Test
public void testBatchUpdate3() {
    NamedParameterJdbcTemplate namedParameterJdbcTemplate =
        new NamedParameterJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(name) values(:myName)";
    UserModel model = new UserModel();
    model.setMyName("name5");
    SqlParameterSource[] params =
        SqlParameterSourceUtils.createBatch(new Object[] {model, model});
    namedParameterJdbcTemplate.batchUpdate(insertSql, params);
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

通过 batchUpdate(String sql, SqlParameterSource[] batchArgs)方法进行命名参数批处理，batchArgs 指定批处理数据集。SqlParameterSourceUtils.createBatch 用于根据 JavaBean 对象或者 Map 创建相应的 BeanPropertySqlParameterSource 或 MapSqlParameterSource。

3) SimpleJdbcTemplate 批处理：已更简单的方式进行批处理；

```
@Test
public void testBatchUpdate4() {
    SimpleJdbcTemplate simpleJdbcTemplate =
        new SimpleJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(name) values(?)";
    List<Object[]> params = new ArrayList<Object[]>();
    params.add(new Object[]{"name5"});
    params.add(new Object[]{"name5"});
    simpleJdbcTemplate.batchUpdate(insertSql, params);
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

本示例使用 batchUpdate(String sql, List<Object[]> batchArgs)方法完成占位符批处理，当然也支持命名参数批处理等。

4）SimpleJdbcInsert 批处理：

```
@Test
public void testBatchUpdate5() {
    SimpleJdbcInsert insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    Map<String, Object> valueMap = new HashMap<String, Object>();
    valueMap.put("name", "name5");
    insert.executeBatch(new Map[] {valueMap, valueMap});
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

如代码所示，使用 executeBatch(Map<String, Object>[] batch)方法执行批处理。

7.5 集成 Spring JDBC 及最佳实践

大多数情况下 Spring JDBC 都是与 IOC 容器一起使用。通过配置方式使用 Spring JDBC。

而且大部分时间都是使用 JdbcTemplate 类（或 SimpleJdbcTemplate 和 NamedParameterJdbcTemplate）进行开发，即可能 80%时间使用 JdbcTemplate 类，而只有 20%时间使用其他类开发，符合 **80/20 法则。**

Spring JDBC 通过实现 DaoSupport 来支持一致的数据库访问。

Spring JDBC 提供如下 DaoSupport 实现：

- **JdbcDaoSupport：**用于支持一致的 JdbcTemplate 访问；
- **NamedParameterJdbcDaoSupport:** 继承 JdbcDaoSupport，同时提供 NamedParameterJdbcTemplate 访问；
- **SimpleJdbcDaoSupport：**继承 JdbcDaoSupport，同时提供 SimpleJdbcTemplate 访问。

由于 JdbcTemplate、NamedParameterJdbcTemplate、SimpleJdbcTemplate 类使用 DataSourceUtils 获取及释放连接，而且连接是与线程绑定的，因此这些 JDBC 模板类是线程安全的，即 JdbcTemplate 对象可以在多线程中重用。

接下来看一下 Spring JDBC 框架的最佳实践：

1)首先定义 Dao 接口

```
package cn.javass.spring.chapter7.dao;
import cn.javass.spring.chapter7.UserModel;
public interface IUserDao {
    public void save(UserModel model);
    public int countAll();
}
```

2）定义 Dao 实现，此处是使用 Spring JDBC 实现：

```
package cn.javass.spring.chapter7.dao.jdbc;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
import cn.javass.spring.chapter7.UserModel;
import cn.javass.spring.chapter7.dao.IUserDao;
public class UserJdbcDaoImpl extends SimpleJdbcDaoSupport implements IUserDao {
   private static final String INSERT_SQL = "insert into test(name) values(:myName)";
   private static final String COUNT_ALL_SQL = "select count(*) from test";

   @Override
   public void save(UserModel model) {
        getSimpleJdbcTemplate().update(INSERT_SQL,
            new BeanPropertySqlParameterSource(model));
   }
   @Override
   public int countAll() {
        return getJdbcTemplate().queryForInt(COUNT_ALL_SQL);
   }
}
```

此处注意首先 Spring JDBC 实现放在 dao.jdbc 包里，如果有 hibernate 实现就放在 dao.hibernate 包里；其次实现类命名如 UserJdbcDaoImpl，即×××JdbcDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

3）进行资源配置（resources/chapter7/applicationContext-resources.xml）：

```
<bean
   class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
      <property name="locations">
         <list>
            <value>classpath:chapter7/resources.properties</value>
         </list>
      </property>
</bean>
```

PropertyPlaceholderConfigurer 用于替换配置元数据，如本示例中将对 bean 定义中的${…}占位符资源用“classpath:chapter7/resources.properties”中相应的元素替换。

```
<bean id="dataSource"
   class="org.springframework.jdbc.datasource.LazyConnectionDataSourceProxy">
      <property name="targetDataSource">
         <bean class="org.logicalcobwebs.proxool.ProxoolDataSource">
               <property name="driver" value="${db.driver.class}" />
               <property name="driverUrl" value="${db.url}" />
               <property name="user" value="${db.username}" />
               <property name="password" value="${db.password}" />
               <property name="maximumConnectionCount"
                        value="${proxool.maxConnCount}" />
               <property name="minimumConnectionCount"
                        value="${proxool.minConnCount}" />
               <property name="statistics" value="${proxool.statistics}" />
               <property name="simultaneousBuildThrottle"
                        value="${proxool.simultaneousBuildThrottle}" />
               <property name="trace" value="${proxool.trace}" />
         </bean>
      </property>
</bean>
```

dataSource 定义数据源，本示例使用 proxool 数据库连接池，并使用 LazyConnectionDataSourceProxy 包装它，从而延迟获取数据库连接；${db.driver.class}将被“classpath:chapter7/resources.properties”中的“db.driver.class”元素属性值替换。

proxool 数据库连接池：本示例使用 proxool-0.9.1 版本，请到 proxool 官网下载并添加 proxool-0.9.1.jar 和 proxool-cglib.jar 到类路径。

ProxoolDataSource 属性含义如下：

- driver：指定数据库驱动；
- driverUrl：数据库连接；
- username：用户名；
- password：密码；
- maximumConnectionCount：连接池最大连接数量；
- minimumConnectionCount：连接池最小连接数量；
- statistics：连接池使用样本状况统计；如 1m,15m,1h,1d 表示没 1 分钟、15 分钟、1 小时及 1 天进行一次样本统计；
- simultaneousBuildThrottle：一次可以创建连接的最大数量；
- trace：true 表示被执行的每个 sql 都将被记录（DEBUG 级别时被打印到相应的日志文件）；

4）定义资源文件（classpath:chapter7/resources.properties）：

```
proxool.maxConnCount=10
proxool.minConnCount=5
proxool.statistics=1m,15m,1h,1d
proxool.simultaneousBuildThrottle=30
proxool.trace=false
db.driver.class=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:mem:test
db.username=sa
db.password=
```

用于替换配置元数据中相应的占位符数据，如${db.driver.class}将被替换为“org.hsqldb.jdbcDriver”。

5）dao 定义配置（chapter7/applicationContext-jdbc.xml）：

```
<bean id="abstractDao" abstract="true">
    <property name="dataSource" ref="dataSource"/>
</bean>
<bean id="userDao"
     class="cn.javass.spring.chapter7.dao.jdbc.UserJdbcDaoImpl"
     parent="abstractDao"/>
```

首先定义抽象的 abstractDao，其有一个 dataSource 属性，从而可以让继承的子类自动继承 dataSource 属性注入；然后定义 userDao，且继承 abstractDao，从而继承 dataSource 注入；我们在此给配置文件命名为 applicationContext-jdbc.xml 表示 Spring JDBC DAO 实现；如果使用 hibernate 实现可以给配置文件命名为 applicationContext-hibernate.xml。

7） 最后测试一下吧（cn.javass.spring.chapter7. JdbcTemplateTest）：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter7/applicationContext-jdbc.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
     Assert.assertEquals(1, userDao.countAll());
}
```

首先读取配置文件，获取 IUserDao 接口实现，然后再调用 IUserDao 接口方法，进行数据库操作，这样对于开发人员使用来说，只面向接口，不关心实现，因此很容易更换实现，比如像更换为 hibernate 实现非常简单。

第八章　对 ORM 的支持

第九章　Spring 的事务

1.1.2　Spring 对 DAO 的支持

第十章　集成其它 Web 框架

第十一章　SSH 集成开发

第十二章　零配置

第十三章　测试

第十四章　IoC 扩展

第十五章　Spring 的 AOP 的接口

第十六章　深入理解 Spring 的 AOP

第十七章　校验和数据绑定

第十八章　Spring 的 MVC

第十九章　综合示例

第二十章　集成视图技术

第二十一章　远程访问

第二十二章　集成 EJB

第二十三章　集成 Web 服务

第二十四章　集成 JMS

第二十五章　集成 JavaMail

第二十六章　集成 JMX

第二十七章　定时调度和线程池

第二十八章　集成动态语言