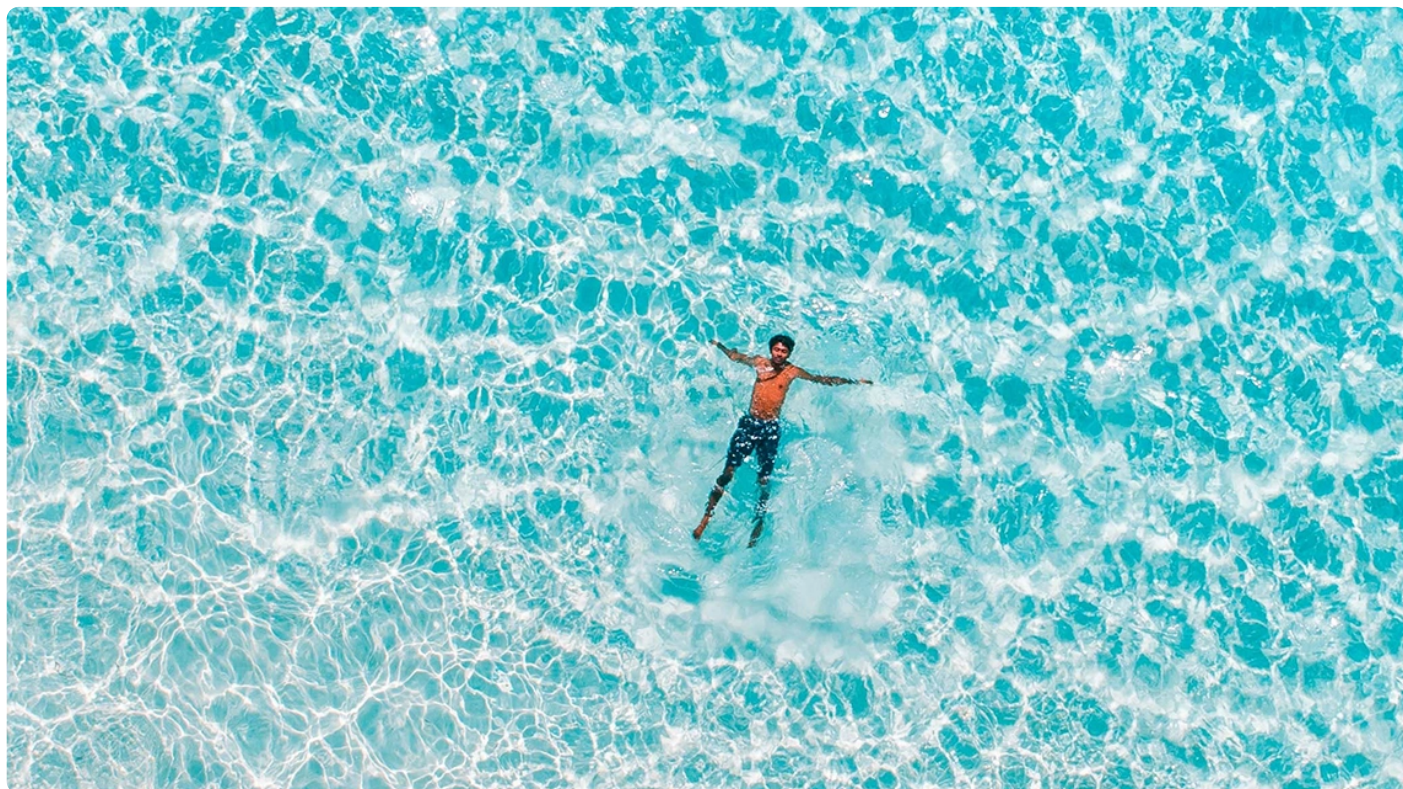


## 14 | 增强模板：如何抽取专门的部件完成专门的任务？

2023-04-12 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。

上节课，我们从 JDBC 这些套路性的程序流程中抽取出了一个通用模板。然后进行了拆解，将 SQL 语句当作参数传入，而 SQL 语句执行之后的结果处理逻辑也作为一个匿名类传入，又抽取出了数据源的概念。下面我们接着上节课的思路，继续拆解 JDBC 程序。


我们现在观察应用程序怎么使用的 JdbcTemplate，看这些代码，还是会发现几个问题。

1. SQL 语句参数的传入还是一个一个写进去的，没有抽取出一个独立的部件进行统一处理。
2. 返回的记录是单行的，不支持多行的数据集，所以能对上层应用程序提供的 API 非常有限。
3. 另外每次执行 SQL 语句都会建立连接、关闭连接，性能会受到很大影响。

这些问题，我们都需要在这节课上一个个解决。


## 参数传入

先看 SQL 语句参数的传入问题，我们注意到现在往 PreparedStatement 中传入参数是这样实现的。

 复制代码

```
1  for (int i = 0; i < args.length; i++) {
2      Object arg = args[i];
3      if (arg instanceof String) {
4          pstmt.setString(i+1, (String)arg);
5      }
6      else if (arg instanceof Integer) {
7          pstmt.setInt(i+1, (int)arg);
8      }
9      else if (arg instanceof java.util.Date) {
10         pstmt.setDate(i+1, new java.sql.Date(((java.util.Date)arg).getTime()));
11     }
12 }
```

简单地说，这些参数都是一个个手工传入进去的。但我们想让参数传入的过程自动化一点，所以现在我们来修改一下，把 JDBC 里传参数的代码进行包装，用一个专门的部件专门做这件事情，于是我们引入 **ArgumentPreparedStatementSetter**，通过里面的 `setValues()` 方法把参数传进 PreparedStatement。

 复制代码

```
1  package com.minis.jdbc.core;
2
3  import java.sql.PreparedStatement;
4  import java.sql.SQLException;
5
6  public class ArgumentPreparedStatementSetter {
7      private final Object[] args; //参数数组
8
9      public ArgumentPreparedStatementSetter(Object[] args) {
10         this.args = args;
11     }
12     //设置SQL参数
13     public void setValues(PreparedStatement pstmt) throws SQLException {
```

```

14     if (this.args != null) {
15         for (int i = 0; i < this.args.length; i++) {
16             Object arg = this.args[i];
17             doSetValue(pstmt, i + 1, arg);
18         }
19     }
20 }
21 //对某个参数, 设置参数值
22 protected void doSetValue(PreparedStatement pstmt, int parameterPosition, Object
23     Object arg = argValue;
24     //判断参数类型, 调用相应的JDBC set方法
25     if (arg instanceof String) {
26         pstmt.setString(parameterPosition, (String)arg);
27     }
28     else if (arg instanceof Integer) {
29         pstmt.setInt(parameterPosition, (int)arg);
30     }
31     else if (arg instanceof java.util.Date) {
32         pstmt.setDate(parameterPosition, new java.sql.Date(((java.util.Date)arg).ge
33     }
34 }
35 }

```

从代码中可以看到, 核心仍然是 JDBC 的 set 方法, 但是包装成了一个独立部件。现在的示例程序只是针对了 String、Int 和 Date 三种数据类型, 更多的数据类型我们留到后面再扩展。

有了这个专门负责参数传入的 setter 之后, query() 就修改成这个样子。

 复制代码

```

1 public Object query(String sql, Object[] args, PreparedStatementCallback pstmtc
2     Connection con = null;
3     PreparedStatement pstmt = null;
4
5     try {
6         //通过data source拿数据库连接
7         con = dataSource.getConnection();
8
9         pstmt = con.prepareStatement(sql);
10        //通过argumentSetter统一设置参数值
11        ArgumentPreparedStatementSetter argumentSetter = new ArgumentPreparedStatement
12        argumentSetter.setValues(pstmt);
13
14        return pstmtcallback.doInPreparedStatement(pstmt);
15    }

```

```
16     catch (Exception e) {
17         e.printStackTrace();
18     }
19     finally {
20         try {
21             pstmt.close();
22             con.close();
23         } catch (Exception e) {
24             }
25     }
26     return null;
27 }
```

我们可以看到，代码简化了很多，手工写的一大堆设置参数的代码不见了，这就体现了专门的部件做专门的事情的优点。

## 对返回结果的处理

JDBC 来执行 SQL 语句，说起来很简单，就三步，一准备参数，二执行语句，三处理返回结果。准备参数和执行语句这两步我们上面都已经抽取了。接下来我们再优化一下处理返回值的代码，看看能不能提供更多便捷的方法。

我们先看一下现在是怎么处理的，程序体现在

`pstmtcallback.doInPreparedStatement(pstmt)` 这个方法里，这是一个 callback 类，由用户程序自己给定，一般会这么做。

[复制代码](#)

```
1  return (User) jdbcTemplate.query(sql, new Object[]{new Integer(userid)},
2      (pstmt)->{
3          ResultSet rs = pstmt.executeQuery();
4          User rtnUser = null;
5          if (rs.next()) {
6              rtnUser = new User();
7              rtnUser.setId(userid);
8              rtnUser.setName(rs.getString("name"));
9              rtnUser.setBirthday(new java.util.Date(rs.getDate("birthday").getTime()))
10         } else {
11         }
12         return rtnUser;
13     }
14 );
```

这个本身没有什么问题，这部分逻辑实际上已经剥离出去了。只不过，它限定了用户只能用这么一种方式进行。有时候很不便利，我们还应该考虑给用户程序提供多种方式。比如说，我们想返回的不是一个对象（对应数据库中一条记录），而是对象列表（对应数据库中多条记录）。这种场景很常见，需要我们再单独提供一个便利的工具。

所以我们设计一个接口 `RowMapper`，把 JDBC 返回的 `ResultSet` 里的某一行数据映射成一个对象。

[复制代码](#)

```
1  package com.minis.jdbc.core;
2
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5
6  public interface RowMapper<T> {
7      T mapRow(ResultSet rs, int rowNum) throws SQLException;
8  }
```

再提供一个接口 `ResultSetExtractor`，把 JDBC 返回的 `ResultSet` 数据集映射为一个集合对象。



```
1 package com.minis.jdbc.core;
2
3 import java.sql.ResultSet;
4 import java.sql.SQLException;
5
6 public interface ResultSetExtractor<T> {
7     T extractData(ResultSet rs) throws SQLException;
8 }
```

[📄 复制代码](#)

利用上面的两个接口，我们来实现一个 RowMapperResultSetExtractor。


```
1 package com.minis.jdbc.core;
2
3 import java.sql.ResultSet;
4 import java.sql.SQLException;
5 import java.util.ArrayList;
6 import java.util.List;
7
8 public class RowMapperResultSetExtractor<T> implements ResultSetExtractor<List<T>>
9     private final RowMapper<T> rowMapper;
10
11     public RowMapperResultSetExtractor(RowMapper<T> rowMapper) {
12         this.rowMapper = rowMapper;
13     }
14
15     @Override
16     public List<T> extractData(ResultSet rs) throws SQLException {
17         List<T> results = new ArrayList<>();
18         int rowNum = 0;
19         //对结果集，循环调用mapRow进行数据记录映射
20         while (rs.next()) {
21             results.add(this.rowMapper.mapRow(rs, rowNum++));
22         }
23         return results;
24     }
25 }
```

[📄 复制代码](#)

这样，SQL 语句返回的数据集就自动映射成对象列表了。我们看到，实际的数据映射工作其实不是我们实现的，而是由 RowMapper 实现的，这个 RowMapper 既是作为一个参数又是

作为一个用户程序传进去的。这很合理，因为确实只有用户程序自己知道数据要如何映射。

好，有了这个工具，我们可以提供一个新的 `query()` 方法来返回 SQL 语句的结果集，代码如下：

 复制代码

```
1  public <T> List<T> query(String sql, Object[] args, RowMapper<T> rowMapper) {
2      RowMapperResultSetExtractor<T> resultExtractor = new RowMapperResultSetExtrac
3      Connection con = null;
4      PreparedStatement pstmt = null;
5      ResultSet rs = null;
6
7      try {
8          //建立数据库连接
9          con = dataSource.getConnection();
10
11         //准备SQL命令语句
12         pstmt = con.prepareStatement(sql);
13         //设置参数
14         ArgumentPreparedStatementSetter argumentSetter = new ArgumentPreparedStatementSetter
15         argumentSetter.setValues(pstmt);
16         //执行语句
17         rs = pstmt.executeQuery();
18
19         //数据库结果集映射为对象列表，返回
20         return resultExtractor.extractData(rs);
21     }
22     catch (Exception e) {
23         e.printStackTrace();
24     }
25     finally {
26         try {
27             pstmt.close();
28             con.close();
29         } catch (Exception e) {
30         }
31     }
32     return null;
33 }
```

那么上层应用程序的 `service` 层要改成这样：

```

1  public List<User> getUsers(int userid) {
2      final String sql = "select id, name,birthday from users where id=?";
3      return (List<User>)jdbcTemplate.query(sql, new Object[]{new Integer(userid)},
4          new RowMapper<User>(){
5              public User mapRow(ResultSet rs, int i) throws SQLException {
6                  User rtnUser = new User();
7                  rtnUser.setId(rs.getInt("id"));
8                  rtnUser.setName(rs.getString("name"));
9                  rtnUser.setBirthday(new java.util.Date(rs.getDate("birthday").getTime
10
11                  return rtnUser;
12              }
13          }
14      );
15  }

```

service 程序里面执行 SQL 语句，直接按照数据记录的字段的 mapping 关系，返回一个对象列表。这样，到此为止，MiniSpring 的 JdbcTemplate 就可以提供 3 种 query() 方法了。

1. public Object query(StatementCallback stmtcallback) {}
2. public Object query(String sql, Object[] args, PreparedStatementCallback pstmtcallback) {}
3. public List query(String sql, Object[] args, RowMapper rowMapper) {}


实际上我们还可以提供更多的工具，你可以举一反三思考一下应该怎么做，这里我就不多说了。

## 数据库连接池

到现在这一步，我们的 MiniSpring 仍然是在执行 SQL 语句的时候，去新建数据库连接，使用完之后就释放掉了。我们知道，数据库连接的建立和释放，是很费资源和时间的。所以这个方案不是最优的，那怎样才能解决这个问题呢？有一个方案可以试一试，那就是**池化技术**。提前在一个池子里预制多个数据库连接，在应用程序来访问的时候，就给它一个，用完之后再收回到池子中，整个过程中数据库连接一直保持不关闭，这样就大大提升了性能。



所以我们需要改造一下原有的数据库连接，不把它真正关闭，而是设置一个可用不可用的标志。我们用一个新的类，叫 PooledConnection，来实现 Connection 接口，里面包含了一个普通的 Connection，然后用一个标志 Active 表示是否可用，并且永不关闭。


 复制代码

```
1 package com.minis.jdbc.pool;
2 public class PooledConnection implements Connection{
3     private Connection connection;
4     private boolean active;
5
6     public PooledConnection() {
7     }
8     public PooledConnection(Connection connection, boolean active) {
9         this.connection = connection;
10        this.active = active;
11    }
12
13    public Connection getConnection() {
14        return connection;
15    }
16    public void setConnection(Connection connection) {
17        this.connection = connection;
18    }
19    public boolean isActive() {
20        return active;
21    }
22    public void setActive(boolean active) {
23        this.active = active;
24    }
25    public void close() throws SQLException {
26        this.active = false;
27    }
28    @Override
29    public PreparedStatement prepareStatement(String sql) throws SQLException {
30        return this.connection.prepareStatement(sql);
31    }
32 }
```

实际代码很长，因为要实现 JDBC Connection 接口里所有的方法，你可以参考上面的示例代码，别的可以都留空。


最主要的，我们要注意 close() 方法，它其实不会关闭连接，只是把这个标志设置为 false。

基于上面的 PooledConnection，我们把原有的 DataSource 改成 PooledDataSource。首先在初始化的时候，就激活所有的数据库连接。

 复制代码


```
1 package com.minis.jdbc.pool;
2
3 public class PooledDataSource implements DataSource{
4     private List<PooledConnection> connections = null;
5     private String driverClassName;
6     private String url;
7     private String username;
8     private String password;
9     private int initialSize = 2;
10    private Properties connectionProperties;
11
12    private void initPool() {
13        this.connections = new ArrayList<>(initialSize);
14        for(int i = 0; i < initialSize; i++){
15            Connection connect = DriverManager.getConnection(url, username, password);
16            PooledConnection pooledConnection = new PooledConnection(connect, false);
17            this.connections.add(pooledConnection);
18        }
19    }
20 }
```

获取数据库连接的代码如下：

 复制代码

```
1 PooledConnection pooledConnection= getAvailableConnection();
2 while(pooledConnection == null){
3     pooledConnection = getAvailableConnection();
4     if(pooledConnection == null){
5         try {
6             TimeUnit.MILLISECONDS.sleep(30);
7         } catch (InterruptedException e) {
8             e.printStackTrace();
9         }
10    }
11 }
12 return pooledConnection;
```


可以看出，我们的策略是死等这一个有效的连接。而获取有效连接的代码如下：

 复制代码

```
1 private PooledConnection getAvailableConnection() throws SQLException{
2     for(PooledConnection pooledConnection : this.connections){
3         if (!pooledConnection.isActive()){
4             pooledConnection.setActive(true);
5             return pooledConnection;
6         }
7     }
8
9     return null;
10 }
```

通过代码可以知道，其实它就是拿一个空闲标志的数据库连接来返回。逻辑上这样是可以的，但是，这段代码就会有一个并发问题，多线程的时候不好用，需要改造一下才能适应多线程环境。我们注意到这个池子用的是一个简单的 ArrayList，这个默认是不同步的，我们需要手工来做同步，比如使用 Collections.synchronizedList()，或者用两个 LinkedBlockingQueue，一个用于 active 连接，一个用于 inactive 连接。

同样，对 DataSource 里数据库的相关信息，可以通过配置来注入的。

 复制代码

```
1 <bean id="dataSource" class="com.minis.jdbc.pool.PooledDataSource">
2     <property name="url" value="jdbc:sqlserver://localhost:1433;databasename=DEMO
3     <property name="driverClassName" value="com.microsoft.sqlserver.jdbc.SQLServe
4     <property name="username" value="sa"/>
5     <property name="password" value="Sql2016"/>
6     <property type="int" name="initialSize" value="3"/>
7 </bean>
```

整个程序的结构实际上没有什么改动，只是将 DataSource 的实现变成了支持连接池的实现。从这里也可以看出，独立抽取部件、解耦这些手段给程序结构带来了极大的灵活性。

## 小结

我们这节课，在已有的 JdbcTemplate 基础之上，仍然按照专门的事情交给专门的部件来做的思路，一步步拆解。

我们把 SQL 语句参数的处理独立成一个 ArgumentPreparedStatementSetter，由它来负责参数的传入。之后对返回结果，我们提供了 RowMapper 和 RowMapperResultSetExtractor，将数据库记录集转换成一个对象的列表，便利了上层应用程序。最后考虑到性能，我们还引入了一个简单的数据库连接池。在这一步步地拆解过程中，JdbcTemplate 这个工具越来越完整、便利了。

完整源代码参见 [🔗 https://github.com/YaleGuo/minis](https://github.com/YaleGuo/minis)。

## 课后题

学完这节课的内容，我也给你留一道思考题。你想一想我们应该怎么改造数据库连接池，保证多线程安全？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)



风轻扬

2023-05-16 来自北京

老师，请教一个问题。query方法中，实例化了一个RowMapperResultSetExtractor，直接用的实现类，没有用接口。而且，我想了一下，也不能用接口，因为Extractor接口里的方法extractData，返回值泛型是Object类型，没办法处理List类型。所以只能是返回list时，new出处理List的Extractor，返回object时，new出处理单对象的Extractor。有点不明白，定义Extractor接口的作用是什么？

作者回复：这是面向接口编程，可以有不同实现，在Spring中其实就不只有一个extractor，只是MiniSpring中仅仅实现了一种。Spring中用了内部类QueryStatementCallback，接收ResultSetExtractor接口，在里面回调rse.extractData(rs)。



**peter**

2023-04-13 来自北京

请教老师几个问题：

Q1：数据库连接池一般设置多大？连接池大小一般是怎么计算的？

Q2：数据库连接池与特定的数据库绑定吗？比如某个连接池可以连接mysql，能连接其他数据库吗？

Q3：常见的数据库连接池有哪些？

Q4：数据库连接池与高并发有什么关系？

作者回复：pool size与具体场景有关，看并发中需要访问数据库的连接数。我看到几个中等规模的系统(10000人以内)，并发量大约是1000-2000，设置的数据库连接池是100-200。

连接池不绑死数据库系统，数据库系统是由driver来支持的。使用Tomcat的话，里面就有很好的连接池。



**马儿**

2023-04-13 来自四川

1. initialPool在第一次getConnection的时候初始化，那么就会存在线程安全问题，可以在方法上粗暴的加一个synchronized并在方法中初始化前提判断是否为空，这样就可以防止连接池中的connections被多次初始化。

2. 获取连接的时候在没有设置active为true之前两个获取连接的线程同时通过了isActive的判断导致两个线程获取到了同一个链接。这里也可以用synchronized来修饰isActive和setActive两个方法，保证一次只有一个线程访问其中的一个方法，不会有两个线程同时访问。并且，对于多核机器来说线程A可能更新完active字段就释放锁了，但是更新后的值还存在自己线程所在的cpu高速缓存中还没有写回到内存，导致线程B读到的还是内存中旧的active值，所以可以再用volatile保证active值修改后马上写回内存并且别的线程也只能从内存读取。

当然，也可以active字段换成线程安全的AtomicBoolean类。

以上是自己的一些思考，请老师指正一下~

作者回复：后面有再回首章节，里面给出了参考方案，你看一下。



