文章编号:1001 - 9081(2006)09 - 2222 - 03

基于模板方法的 JDBC API 的抽象封装与实现

刘　泉,赵晓明
(武汉理工大学 信息工程学院, 湖北 武汉 430070)
(shamine@163.com)

摘　要:作为访问关系数据库的核心标准,JDBC 在 Java 平台的应用上发挥着至关重要的作用,然而它自身的不足增加了直接基于 JDBC API 应用的复杂度。分析了直接采用 JDBC API 访问数据库的相关问题,并利用模板方法和回调函数对 API 进行抽象封装,且给出了相应的实现,最后比较了直接采用 JDBC API 和对 API 进行封装后的性能差异,证明该封装与实现是一个良好的实践模型。

关键词:Java 数据库连接; Java; 模板方法; 回调函数
中图分类号: TP311.11　　**文献标识码**:A

Wrapping JDBC API with template method pattern

LIU Quan, ZHAO Xiao-ming
(*School of Information Engineering, Wuhan University of Technology, Wuhan Hubei* 430070, *China*)

Abstract: As a standard core specification to access relational database, JDBC(Java Database Connectivity) plays an important role in the Java platform applications. However, due to its own deficiency, complexities are introduced into the applications directly based on JDBC API(Application Programming Interface). Some related issues when using JDBC API to access database were discussed, and template method and callback functions were adopted to wrap and implement API. Then, the performances between using JDBC API directly and using wrapped API were compared. It is proved that the latter approach has better performance in application.

Key words: JDBC(Java Database Connectivity); Java; template method; callback function

0　引言

JDBC 是操作关系数据库的标准访问协议,提供了 Java 应用程序访问数据库的标准方法[1]。其标准化的 API 为基于 Java 的数据访问应用程序的可移植性奠定了基础,为各种数据的访问提供了跨数据库的支持,是标准 Java(J2SE)和企业 Java(J2EE)中一个不可缺少的重要组成部分。

但单纯地使用 JDBC API 来构建应用程序,代码的冗余度会非常高,编程效率低下,而且容易出错,不利于程序的维护和扩展。如果运用有效的方式对其进行抽象封装,在这个抽象层次上进行开发,就会大大提高程序的灵活性、重用性和可靠性。

1　使用 JDBC API 访问数据库

1.1　正确释放数据库资源

JDBC 应用中的一个常见错误是没有正确关闭连接(Connection)。这将导致数据库资源的不合理分配。类似地,关闭语句(Statement)和结果集(ResultSet)也通常是被推荐的操作[2]。

释放数据库资源的操作是一件费心的事情。即使是有经验的程序员,也常常发生因关注应用逻辑而忘记释放数据库连接的事情。在一个使用了数据库连接池的应用中,没有及时释放连接将会导致连接池溢出。由于数据库资源被占用,连接池只好分配一个新的连接供程序使用。每当这些有缺陷的程序片断要求连接池提供访问资源时,就将积累一个无效的连接。无效连接的数目达到连接池的最大容量,再次访问时就会发生连接池溢出的异常。而这种错误通常则是经历了很长时间之后才会出现,回头检查程序缺陷费时费力,很难找到是什么地方出现的问题。

1.2　丑陋的代码

使用 JDBC API 操作数据库,需要完成以下步骤:1)加载 JDBC 驱动; 2)建立连接; 3)建立用于查询或更新的语句; 4)处理结果; 5)关闭连接[3]。繁琐的代码增加了程序员的工作,也影响了代码的阅读和后期的维护。

一般来说,直接使用 JDBC API 操作数据库的方式如清单 1 所示。

清单 1:传统 JDBC 访问模式

```
String sql = "SELECT NAME, AGE FROM author";
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
 /*取得 JDBC Connection, Statement, ResultSet 对象,进行相应
    的操作*/
} catch (SQLException e) {
   // 处理异常情况
} finally { // 关闭资源
   if (rs != null)
   try {
    rs.close();
   } catch (SQLException e) {
      // 处理异常
   }
   if (stmt != null)
   try {
    stmt.close();
   } catch (SQLException e) {
      // 处理异常
   }
   if (conn != null)
```

收稿日期:2006 - 03 - 10; **修订日期**:2006 - 05 - 31　　**基金项目**:国家科技攻关资助项目(20041g0113)
作者简介:刘泉(1963 -),女,湖北武汉人,教授,博士生导师,主要研究方向:计算机网络通信、信号处理、非线性系统; 赵晓明(1981 -),男,湖北荆门人,硕士研究生,主要研究方向:网格、SOA.

```
    try {
      conn. close( ) ;
    } catch ( SQLException e) {
      // 处理异常
    }
}
```

其中 finally 字句通常被用来确保关闭数据库数据集、语句和连接。代码是冗长、膨胀和重复的。大部分的冗余代码所要做的都是一些常规操作,比如数据库资源的获得与释放,异常的捕获和处理等。直接采用 JDBC API,程序员的精力无法放到真正所要关注的业务逻辑上。

1.3　难于进行高级应用

直接采用 JDBC,对于某些特殊的高级应用显得相当困难,甚至是无法执行。因为数据库资源的获得、操作和关闭是一系列的操作,缺一不可。倘若在获得数据库资源之后,需要进行一系列其他的费时的操作,那么这个被占用的数据库资源将无法被程序的其他部分使用,直到资源被释放为止。

一个直观的例子是数据缓存。有时候从数据库取得的数据将要缓存数个小时,甚至更长。若缓存的对象是 ResultSet,那么意味着相关的 Statement 和 Connection 也要被占用数个小时,不但程序的接口需要精心设计,以便释放 ResultSet 的时候能访问到相关的 Statement 和 Connection 对象,而且要保证参数传递的过程中其他代码块不得对这些资源进行误操作。程序的健壮性难于保证,在中、大型的项目中多半是一场灾难。

2　用模板方法与回调函数封装 JDBC API

2.1　模板方法

模板方法(Template Method)模式是 GoF 提出的 23 种常用设计模式中的一种。模板方法定义了一个操作算法中的骨架,而将一些步骤延迟到子类中,使得子类可以不改变一个算法的结构即可重新定义该算法的某些特定步骤[4]。结构模型如图 1 所示。

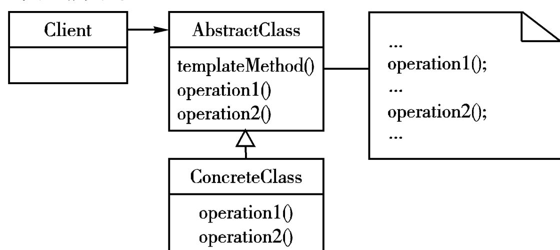

图 1　结构模型

对于直接基于 JDBC API 的应用来说,操作数据库具有一致的行为,其中资源的管理和异常的处理则是操作数据库的算法骨架中不变的部分,而数据的处理则根据具体的要求而改变。不变的部分定义到模板方法中,而变化的部分则作为抽象方法延迟到子类中执行。

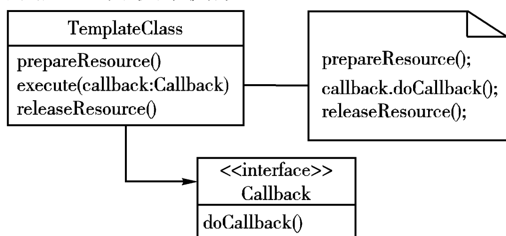

图 2　重构后的模型

为了不产生大量的继承,使程序结构更为清晰明了,我们对模板方法模式进行重构,采用回调来处理原来模板方法中需要延迟到子类中的抽象方法。重构后的模型如图 2 所示。

这样,抽象类变成了具体的模板类。模板方法 execute (Callback)定义了算法结构,可变的算法逻辑通过回调接口 Callback 定义的方法 doCallback( )调用,依赖于接口的具体实现。根据应用,需要对 Connection、Statement 和 ResultSet 进行三种形式的回调。

2.2　Connection 和 Statement 的回调

Connection 和 Statement 的回调处理比较类似。以 Statement 为例,客户端的代码如清单 2 所示。

清单 2:Statement 回调模式的客户端代码

```
JdbcUtils ju = new JdbcUtils( getDataSource( ) ) ;
final String sql = "INSERT INTO author SET name = 'liu'";
ju . update( new StatementCallback( ) {
  public Object doInStatement( Statement stmt)
  throws SQLException {
    return new Integer( stmt. executeUpdate( sql) ) ;
  }
}) ;
```

可见,使用匿名内部类的形式实现回调接口,处理 JDBC 的应用要比清单 1 所列出的代码简捷有效的多。

我们着重考察的对象是 StatementCallback 回调接口和 update(StatementCallback) 模板方法。StatementCallback 接口定义了共有方法 doInStatement(Statement),我们需要具体实现该方法处理过程,这里是向数据库添加一条记录,而数据库资源的获取和释放,异常处理等其他工作都交由模板方法来完成。清单 3 列出了模板方法的具体处理过程。

清单 3:Statement 回调模式的具体处理过程

```
Connection conn = null;
Statement stmt = null;
try {
  // 获取数据库资源
  conn = getDataSource( ) . getConnection( ) ;
  stmt = conn. createStatement( ) ;
  // 执行接口方法,处理具体的业务逻辑
  return action. doInStatement( stmt) ;
} catch ( SQLException e) {
    // 处理异常
    throw new JdbcRuntimeException( e) ;
} finally {
   // 释放资源
   release( null, stmt, conn) ;
}
```

实现 Connection 的处理方式与 Statement 的处理很相似,回调接口 ConnectionCallback,模板方法 execute (ConnectionCallback),具体代码清单不再赘述。

2.3　ResultSet 的回调

ResultSet 的回调有点特殊,它不像 Statement 的处理,SQL 语句执行完就可以释放资源了,它需要返回查询的 ResultSet 对象待用。一般来说,ResultSet 的使用可能延迟执行,也可能需要保留一段时间后再进行处理,例如数据缓存。为了确保 ResultSet 及时关闭,可以把它转换成为其他的数据形式返回。类似地,通过实现 ResultSetCallback 回调接口来处理转换的过程。实现的模板方法定义为 query (String, ResultSetCallback)。

事实上,鉴于 ResultSet 记录与记录之间数据格式相同,记录集合为链式列表的数据结构的等特点,我们可以只实现一条数据的转换映射,而记录集合的循环、列表形成等处理都交给模板方法来完成。

针对我们想要查询 Author 对象列表的功能,客户端的代码如清单 4 所示。

清单 4:ResultSet 回调模式的客户端代码

```
JdbcUtils ju = new JdbcUtils(getDataSource());
String sql = "SELECT authorId, name FROM author";
List authors = ju.queryForList(sql, new RowMapper() {
  public Object mapRow(ResultSet rs)
  throws SQLException { // 执行一条记录的转换
    Author author = new Author();
    author.setAuthorId(rs.getInt(1));
    author.setName(rs.getString(2));
    return author;
  }
}));
```

其中接口 RowMapper 的 mapRow(ResultSet)方法是具体的业务逻辑,由回调接口根据具体的实现进行处理。清单 5 列出了模板方法 queryForList(String, RowMapper)的具体处理过程,它本身则实现了 ResultSetCallback 的回调接口:

清单 5:ResultSet 回调模式的具体处理过程

```
public List queryForList(String sql, final RowMapper rowMapper) {
  return (List) query(sql, new ResultSetCallback() {
    // 实现 ResultSetCallback 的回调接口
    public Object doInResultSet(ResultSet rs)
    throws SQLException {
      List results = new ArrayList();
      while (rs.next()) {
        // 执行 RowMapper 的回调方法
        results.add(rowMapper.mapRow(rs));
      }
      return results;
    }
  });
}
```

模板方法 query(String, ResultSetCallback)的实现与清单 3 比较类似,不再赘述。

3 性能差异

testQueryByJU( ) 函数采用清单 4 的算法实现;testQueryByJDBC( )函数采用清单 1 的算法实现,函数返回值为消耗的时间间隔(ms),执行的功能均将 Author 对象封装到 List 中。查询 5000 条记录。

testQueryByJU( )与 testQueryByJDBC( )分别执行 20 次查询,得出查询总消耗时间 $s1$ 和 $s2$ 以及平均消耗时间 $t1$ 和 $t2$;经过 10 次这样的比较后得出两者总时间消耗的比率 r。

利用上述的算法,得出的数据和结果如表 1 所示。

表 1 性能比较

序号	$s1$/ms	$s2$/ms	$t1$/ms	$t2$/ms
1	1091.0	1062.0	54.55	53.10
2	1082.0	1001.0	54.10	50.05
3	1021.0	1032.0	51.05	51.60
4	1031.0	1112.0	51.55	55.60
5	1022.0	1011.0	51.10	50.55
6	1092.0	1011.0	54.60	50.55
7	1021.0	1002.0	51.05	50.10
8	1021.0	1042.0	51.05	52.10
9	981.0	1052.0	49.05	52.60
10	1031.0	1012.0	51.55	50.60

$$r = \frac{\sum s1}{\sum s2} \approx 1.0054$$

根据测试的结果看,封装后的函数性能与直接采用 JDBC API 的性能差异不大,大约只有 5.4‰的损失。

4 结语

利用模板方法,通过回调用户实现的处理接口,共同完成了数据库数据的存取所需要的功能。可根据实际情况创建一系列默认的回调方式和模板方法对其进行扩展,而针对特殊的应用则可实现回调接口来完成。

采用模板方法与回调函数返回的结果与数据库再无关联。虽然相比直接采用 JDBC API 略有性能上的损失,但数据库资源及时的释放可快速响应其他的处理请求,在并发情况下有助于提高数据库访问的性能。

花费微小的性能代价可获得自动的资源管理、一致的编程模型以及良好的灵活性与扩展性,让程序员集中精力处理业务逻辑,提高了工作效率与质量,使得整个应用更加稳定健壮。此外,由于该方案基于标准的 JDBC API,不依赖于其他任何基础架构,例如 EJB 容器等,程序的调用和调试简单,可单独使用,也可集成到其他框架中作为数据库操作的底层接口。

参考文献:

[1] ELLIS J, HO L. JDBC 3.0 Specification[EB/OL]. http://java.sun.com/products/jdbc/download.html, 2001.

[2] MARX D. Add Some Spring to Your Oracle JDBC Access [EB/OL]. http://www.oracle.com/technology/pub/articles/marx_spring.html, 2005.

[3] BROWN S. JSP 编程指南[M]. 北京:电子工业出版社,2002.

[4] GAMMA E, HELM R, JOHNSON R, *et al*. Design Patterns: Elements of Reusable Object-Oriented Software[M]. Boston: Addison-Wesley Professional, 1995.

(上接第 2210 页)

[3] CHIANG T, ZHANG Y. A new rate control scheme using quadratic rate distortion model[J]. IEEE Trans, 1997, CSVT-7(I): 287 - 311.

[4] MA SW, GAO W, LU Y. Rate Control on JVT Standard., Document: JVT-D030[A]. JVT4th Meeting[C]. Klagenfurt, Austria, 2002.

[5] MA SW, GAO W, LU Y, *et al*. Improved Rate Control Algorithm, Document: JVT-E069[A]. JVT5thMeeting[C]. Geneva, CH, 2002.

[6] MA SW, GAO W, LU Y, , *et al*. Proposed draft description of rate control on JVT standard, Document: JVT-F086[A]. JVT6th Meeting[C]. Awaji, 2002.

[7] LI ZG, PAN F, PANG K. Adaptive Basic Unit Layer Rate Control for JVT, Document: JVT-G012[A]. JVT7th Meeting[C]. Pattaya II, Thailand, 2003.

[8] LI ZG, GAO W, PAN F. Adaptive Rate Control with HRD Consideration, Document: JVT-H014[A]. JVT8thMeeting[C]. Geneva, 2003.

[9] MILANI S, CELETTO L, MIAN GA. A Rate Control Algorithm for the H.264 Encoder[EB/OL]. http://primo.ismb.it/firb/docs/milani1.pdf.

[10] 陈川,余松煜. 联合编码模式选择的码率控制算法[J]. 电子学报,2004,32(5).

[11] 李蔷,余松煜. 一种高效的 H.264 码率控制方法[J]. 上海交通大学学报,2004,38(11).

[12] YUAN W, LIN SX, ZHANG YD, *et al*. Optimum Bit Allocation and Rate Control for H.264/AVC, Document: JVT-O016[A]. JVT 15th Meeting[C]. Busan, KR, 2005.

[13] JIANG MQ, YI XQ, LING N. Improved Frame-Layer Rate Control for H.264 Using MADRatio[EB/OL]. http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1328871.

[14] YI XQ, LING N. Rate Control Using Enhanced Frame Complexity Measure For H.264 Video[EB/OL]. http://students.engr.scu.edu/~xyi/publication/SIPS2004_RC.pdf.