持久化工具：Memory 介绍



# 1 概述

## 1.1 连接、语句和结果集

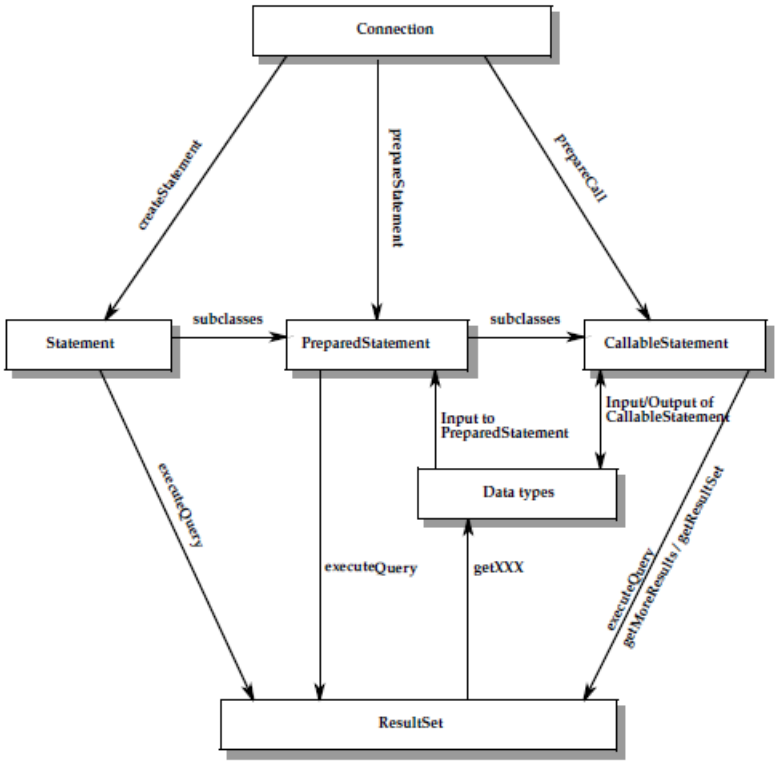


FIGURE 5-1 Relationships between major classes and interface in the `java.sql` package

http://download.oracle.com/otndocs/jcp/jdbc-4_1-mrel-spec/index.html

从 JDBC 的规范上看，其对数据访问层有相当简洁的抽象：1、连接(connection) 2、语句(statement)、3 结果集(result set)。我们对数据库做的事情无非：连接数据库，执行语句，拿到结果。

因此，持久化的工具的目的就不言自明了：进一步简化连接的管理、语句的执行、结果集提取等操作。下面从获取结果集、管理连接、语句预处理等 3 方面逐一阐述工具做了哪些事情。

这里提一句，Memory 在设计与实现上，都借鉴了 Dbutils，其相对于 hibernate, mybatis 这些庞然大物，已经是一个极其小巧的工具。但是 Memory 的类和接口更少（不超过 10 个），体积更小（只有二十几 K），数目和体积都约为 dbutils 的 1/3，却添加了非常实用的功能：

- 将简单的 POJO 对象直接持久化到数据库中；
- 打印运行时出错的 SQL 语句，其可以直接拷贝到数据库客户端上进行调试；
- 直截了当的分页查询。

## 1.2 获取结果集

获取结果集，就是把 ResultSet 转换为目标数据结构，这里使用 T（泛型）泛指各种数据结构。我们定义一个接口类来表示这件事情：

```java
public interface ResultSetHandler<T> {
    T handle(ResultSet rs) throws SQLException;
}
```

在实际应用中，结果集是某张表的一行或多行数据时，常使用 BeanHandler、BeanListHandler 或 JSONObjectHandler、JSONArrayHandler 进行处理，结果集是某一列的一行或多行数据时，使用 ColumnHandler、ColumnListHandler 进行处理。

## 1.3 连接的管理

将连接的交给外部的数据源(DataSource)进行统一管理。比如使用 Tomcat 容器自带的数据源。

在 Tomcat 的 context.xml 文件配置数据源 xxxxx：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Context>
<Resource name="jdbc/xxxxx"
        auth="Container"
        type="javax.sql.DataSource"
        driverClassName="oracle.jdbc.OracleDriver"
        url="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HO
        username=""
        password=""
        maxActive="100"
        maxIdle="30"
        maxWait="3000"
        validationQuery = "SELECT 1 FROM DUAL"
        testWhileIdle = "true"
        testOnBorrow = "true"
        timeBetweenEvictionRunsMillis = "3600000"
        minEvictableIdleTimeMillis = "18000000"
/>
</Context>
```

在代码中实例化（采用懒加载单例模式）该数据源：

```java
public class MemoryFactory {

    private MemoryFactory() {
    }

    private static class SingletonHolder {
        public static final Memory MEMORY = new Memory(new SimpleDataSource());
    }

    public static Memory getInstance() {
        return SingletonHolder.MEMORY;
    }

    /**
     * 在容器(Tomcat)运行状态下，可使用getDataSource()
     */
    public static final DataSource getDataSource() {
        try {
            Context context = new InitialContext();
            return (DataSource) context.lookup("java:comp/env/jdbc/test");
        } catch (NamingException e) {
            throw new RuntimeException(e);
        }
    }
}
```

## 1.4 语句预处理

与 ResultSetHanlder 相互呼应，提供了 PreparedStatementHanlder 类，这个类提供语句（PreparedStatment）一些辅助性的方法，比如生成运行时的 SQL 语句、调整日期格式、简

化分页语句写法等。这个类在应用中不会直接用到。其作用将隐藏在最重要的一个类 Memory 之中（与这个工具命名相同）。

# 2 使用

上章从结果集提取、连接管理、语句处理等 3 个角度介绍了这个工具，本章介绍的 Memory 类就是对 3 者的集成，分 3 节描述 Memory 开放的 API。

## 2.1 命令与查询

对数据库所有的操作，可分为两类：命令与查询。命令即更新数据，可进一步分为新增、删除与编辑。

### 2.1.1 查询(query)

```
public <T> T query(StringBuffer sql, ResultSetHandler<T> rsh,
public <T> T query(String sql, ResultSetHandler<T> rsh, Object... params)
public <T> T query(Connection conn, StringBuffer sql,
public <T> T query(Connection conn, String sql, ResultSetHandler<T> rsh,
```

从接口定义可以看出，查询(query)方法，返回结果集，参数名也相似，只是数据结构不同而已：StringBuffer 和 List 一组，String 和 Array（变长参数）一组，没有传递 Connection 参数，则表明连接在 memory 内部管理；有传递 Connection 参数，则表明连接交给外部程序管理。

在这个层面使用 API，就是写 SQL 语句，几乎没有任何限制，唯一的限制就是在使用 BeanHandler 与 BeanListHandler 时，**Bean 的字段与 Table 的字段要存在相互匹配，Bean 的字段命名风格是驼峰式，Table 的字段命名是下划线连接**。

### 2.1.2 命令(update)

```
public int update(StringBuffer sql, List<Object> params)
public int update(String sql, Object... params) throws SQLException {
public int update(Connection conn, StringBuffer sql, List<Object> params)
public int update(Connection conn, String sql, Object... params)

public int[] batch(String sql, Object[][] params) throws SQLException {
public int[] batch(Connection conn, String sql, Object[][] params) throws SQLException {
```

相对于查询(query)方法，更新(update)方法，没有结果集处理器(ResultSetHandler)的参数以及结果集转化为的对象。

但更新有批量更新(batch)的方法，提供批量执行 sql 语句的功能。

## 2.2 增删改查(CRUD)

增删改查，英文缩写为 CRUD，这个大家都非常熟悉，使用 Create, read, update, delete 来做作为接口名称，这样记忆和理解成本最低。

Lifesinger 在 jQuery 为什么优秀兼谈库与框架的设计一文中，提到：**在类库界，解决了 What，解决了定位问题后，基本上已经决定了生死存亡。** 至于 How，也重要但往往不是关键。https://github.com/lifesinger/lifesinger.github.com/issues/114

本人对此深以为然，所以 Memory 工具在接口方法名称、类名等的使用上相当节制（数量尽量少），这点也不同于别的持久化工具。

### 2.2.1 新增(create)

```
public <T> int create(Class<T> cls, T bean) throws SQLException {
public <T> int create(Class<T> cls, T bean, boolean customKey) throws SQLException {
public <T> int create(Connection conn, Class<T> cls, T bean)
public <T> int create(Connection conn, Class<T> cls, T bean,

public <T> int[] create(Class<T> cls, List<T> beans) throws SQLException {
public <T> int[] create(Class<T> cls, List<T> beans, boolean customKey) throws SQLException {
public <T> int[] create(Connection conn, Class<T> cls, List<T> beans) throws SQLException {
public <T> int[] create(Connection conn, Class<T> cls, List<T> beans, boolean customKey) throws SQLException {
```

这些接口可持久化新增的一个对象或多个对象时。customkey 这个参数表示主键的值是否使用自定的值。如果不是使用自定义的值，则采用序列(oracle)或自增主键(mysql)，此时主键的名称必须是 ID。

### 2.2.2 查询(read)

```
public <T> T read(Class<T> cls, long id) throws SQLException {
public <T> T read(Connection conn, Class<T> cls, long id)
```

根据主键（主键名必须为 ID）读取一条记录，并转化为对象。

### 2.2.3 更新(update)

```
public <T> int update(Class<T> cls, T bean) throws SQLException {
public <T> int update(Connection conn,Class<T> cls, T bean) throws SQLException {
public <T> int update(Class<T> cls, T bean, String primaryKey) throws SQLException {
public <T> int update(Connection conn, Class<T> cls, T bean, String primaryKey)

public <T> int[] update(Class<T> cls, List<T> beans) throws SQLException {
public <T> int[] update(Connection conn, Class<T> cls, List<T> beans) throws SQLException {
public <T> int[] update(Class<T> cls, List<T> beans,String primaryKey) throws SQLException {
public <T> int[] update(Connection conn, Class<T> cls, List<T> beans,String primaryKey) throws SQLException {
```

这些接口可持久化更新的一个对象或多个对象时。primaryKey 这个参数指定主键名称，默认是 ID。

### 2.2.4 删除(delete)

```
public <T> int delete(Class<T> cls, long id) throws SQLException {
public <T> int delete(Connection conn, Class<T> cls, long id)
```

根据主键（主键名必须为 ID）删除一条记录。

## 2.3 其它

Memory 的 API 在 SQL 语句操作层面分为：命令与查询（2.1 节），在对象操作层面分为：增删改查（2.2 节）。查询有一些常用的辅助性操作，比如分页和 IN 语句；在对事务有要求的场合，memory 提供获取连接的接口，并将连接交给应用自行控制。

### 2.3.1 分页

```
public void pager(StringBuffer sql, List<Object> params, int pageSize,
```

分页查询几乎是必不可少的，但是 oracle 的分页查询语句写起来相当复杂（3 重嵌套），mysql 分页查询虽然简单，但是其参数 limit offset, n 也不够直观。

分页查询，即在问如果每页 pageSize 条记录，那么第 pageNo 页的记录是什么。分页查询接口(pager)封装了 oracle 和 mysql 的查询语句，并提供了 pageSize 和 pageNo 两个直观的参数。

### 2.3.2 IN 语句

```
public <T> void in(StringBuffer sql, List<Object> params, String operator,
```

IN 语句在查询时也比较常用，占位符?必须与参数的个数相匹配，手工拼接容易出错；当参数个数是动态变化时，占位符的拼写更是繁琐，因此对 IN 语句做了一个简单的封装，以保持代码的简洁。

### 2.3.3 事务

```
public Connection getConnection() throws SQLException {
```

可以从 memory 取出一条连接，然后设置连接为非自动提交，进行事务操作与回滚。

# 3 多余的废话

## 3.1 为什么不用链式写法？

不少持久化的库或框架，喜欢使用链式写法来写 SQL 语句。但是殊不知链式的写法在

Jquery 很自然，在 SQL 中却是生搬硬套，不得其法。**SQL 是数据库领域的专用语言(DSL)，用其本来的写法来表达是最自然的**。

比如：
http://droidparts.org/orm.html#many-to-many

```
// Select is used to provide data to EntityCursorAdapter
Select<EntityType> select = select().columns("_id", "name").where("external_id", Is.EQUAL, 10);

// alternatively, call execute() to get the underlying Cursor
Cursor cursor = select().where("name", Is.LIKE, "%%alex%%").execute();

// use Where object for complex queries
Where haveCoordinaltes = new Where("latitude", Is.NOT_EQUAL, 0).or("longitude", Is.NOT_EQUAL, 0);
select().where("country", Is.EQUAL, "us").where(haveCoordinates);
```

又比如：

```
Condition c = Cnd.where("age",">",30).and("name", "LIKE", "%K%").asc("name").desc("id");
```

这些库的设计与 Hibernate 的 Criterion 多多少少有些相似，把 SQL 简单明了的写法改成所谓面向对象的链式写法。关系和对象变得扭曲（Object-Relational Impedance Mismatch），让人几乎看不到 SQL 本身的简洁和链式写法（builder pattern）的优雅，一举两“失”。

## 3.2 为什么不用 XML 或 Annotation 配置?

只要我们约定了表名与类名、列名与字段名的命名规则，并严格遵循，何须在再去了解 XML 和 annotation 配置的写法，再去写 XML 和 Annotation 维护映射关系呢？。少了这些额外的东西，代码的可维护性和可读性是不是也大大提高了呢。

试举一些持久化框架的做法：
比如：
http://droidparts.org/orm.html#many-to-many

```
@Table(name="track_to_tag")
public class TrackToTag extends Entity {
    @Column(nullable = false)
    public Track track;
    @Column(nullable = false)
    public Tag tag;
}
```

又比如：
http://www.nutzam.com/core/dao/dynamic_table_name.html

```
@Table("t_company")
public class Company {

    @Id
    private int id;

    @Name
    private String name;

    @Column
    private int ceoId;

    @One(target = Employee.class, field = "ceoId")
    private Employee CEO;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
```

XML 繁琐冗长的配置，比如 Ibatis 或 Hibernate，就不拷贝举例了。

## 3.3 为什么只用 PreparedStatement?

Statement 和 CallableStatement 只在极少的场景，比如复杂的数据导入导出，可能用到。但在绝多大多数场景，PreparedStatment 相对 Statement 更高效、更安全，代码的可读性更好；而 CallableStatment，是把业务逻辑隐藏在 SQL 的存储过程，而不是显化在代码之中，理解代码将变得更困难，可读性也不如 PreparedStatement。

## 3.4 能不能把运行时的 SQL 语句打印出来？

在开发过程，SQL 语句有可能写错，如果能把运行时出错的 SQL 语句直接打印出来，那对排错非常方便，因为其可以直接拷贝到数据库客户端进行调试。

在 https://www.ibm.com/developerworks/cn/java/j-loggable/这篇文章中，作者也希望有一种方法，它使我们能够获得查询字符串，并用实际的参数值替换参数占位符，最终他提出了一种解决方案，使用修饰器模式(decorator)扩展 PreparedStatement，新增一个有日志功能的 LoggableStatment 的类。这当然是很不错的解决方案。

在 memory 工具，没有新增扩展类，只是在 PrparedStatementHandler 中，提供一个 print 方法，将 SQL 语句中的占位符替换为实际的参数，并在发生 SQL Exception 时，将其打印出来。

## 3.5 也说 ORM

在开源中国可以搜到数百个 ORM 框架或类库。可见 ORM 曾经、也许现在还是，让不少攻城狮和程序猿，趋之若鹜。当然也有人对其反思，有一篇文章《为什么我说 ORM 是一种反模式》，就提出不同的看法。

文章的中文链接：

http://www.nowamagic.net/librarys/veda/detail/2217

文章的英文连接：

https://github.com/brettwooldridge/SansOrm/wiki/ORM-is-an-anti-pattern

ORM，通俗讲，就是把一种问题转化为另一种问题进行解决。但是数据库的问题，比如关联查询、分页、排序，能在 OOP 中得以完美的解决吗？OOP 恐怕心有余而力不足。而这些问题却是关系数据库最擅长的问题域。

把关系数据库擅长解决的问题转化给不擅长处理这类问题的 OOP 去解决，这不是很糊涂吗？OOP 的方法论，应当控制一下自己的野心，专注于自己擅长的领域，比如代码的组织与管理、界面开发的应用等等。

当然 ORM 也不是一无是处，把一条数据（结果集）自动转化为一个对象，以便于业务代码的处理还是有益处的。但要把所有的关系操作映射为对象的操作（比如外键关系映射为继承），或者反之（比如将继承映射为外键关系），必定是事倍功半、得不偿失。

# 4 参考文献


http://commons.apache.org/proper/commons-dbutils/

http://www.nowamagic.net/librarys/veda/detail/2217

https://github.com/brettwooldridge/SansOrm/wiki/ORM-is-an-anti-pattern

http://segmentfault.com/a/1190000000378827

http://www.oschina.net/project/tag/126/orm

http://www.nutzam.com/core/dao/annotations.html

https://github.com/lifesinger/lifesinger.github.com/issues/114

https://github.com/brettwooldridge/SansOrm/wiki/ORM-is-an-anti-pattern

http://www.oschina.net/translate/secrets-of-awesome-javascript-api-design

https://github.com/lifesinger/lifesinger.github.com/issues/114_

http://download.oracle.com/otndocs/jcp/jdbc-4_1-mrel-spec/index.html

http://en.wikipedia.org/wiki/Singleton_pattern#Lazy_initialization