课程目标：

Jdbc的优化！

1. BeanUtils组件
2. 自定义一个持久层的框架
3. DbUtils组件
4. 案例优化

# BeanUtils组件

## 简介

程序中对javabean的操作很频繁， 所以apache提供了一套开源的api，方便对javabean的操作！即BeanUtils组件。

BeanUtils组件， 作用是简化javabean的操作！

用户可以从[www.apache.org下载BeanUtils](http://www.apache.org下载BeanUtils)组件，然后再在项目中引入jar文件！

**使用BenUtils组件：**

1. 引入commons-beanutils-1.8.3.jar核心包
2. 引入日志支持包: commons-logging-1.1.3.jar

**如果缺少日志jar文件，报错：**

java.lang.NoClassDefFoundError: org/apache/commons/logging/LogFactory

at org.apache.commons.beanutils.ConvertUtilsBean.<init>(ConvertUtilsBean.java:157)

at org.apache.commons.beanutils.BeanUtilsBean.<init>(BeanUtilsBean.java:117)

at org.apache.commons.beanutils.BeanUtilsBean$1.initialValue(BeanUtilsBean.java:68)

at

## 实例, 基本用法

方法1： 对象属性的拷贝

BeanUtils.copyProperty(admin, "userName", "jack");

BeanUtils.setProperty(admin, "age", 18);

方法2： 对象的拷贝

BeanUtils.copyProperties(newAdmin, admin);

方法3： map数据拷贝到javabean中

【注意：map中的key要与javabean的属性名称一致】

BeanUtils.populate(adminMap, map);

|  |
| --- |
| //1. 对javabean的基本操作  @Test  **public** **void** test1() **throws** Exception {    // a. 基本操作  Admin admin = **new** Admin();  // admin.setUserName("Jack");  // admin.setPwd("999");    // b. BeanUtils组件实现对象属性的拷贝  BeanUtils.*copyProperty*(admin, "userName", "jack");  BeanUtils.*setProperty*(admin, "age", 18);    // 总结1： 对于基本数据类型，会自动进行类型转换!      // c. 对象的拷贝  Admin newAdmin = **new** Admin();  BeanUtils.*copyProperties*(newAdmin, admin);    // d. map数据，拷贝到对象中  Admin adminMap = **new** Admin();  Map<String,Object> map = **new** HashMap<String,Object>();  map.put("userName", "Jerry");  map.put("age", 29);  // 注意：map中的key要与javabean的属性名称一致  BeanUtils.*populate*(adminMap, map);    // 测试  System.*out*.println(adminMap.getUserName());  System.*out*.println(adminMap.getAge());  } |

## 实例, 日期类型的拷贝

需要注册日期类型转换器，2种方式参见下面代码：

|  |
| --- |
| **public** **class** App {  //1. 对javabean的基本操作  @Test  **public** **void** test1() **throws** Exception {    // a. 基本操作  Admin admin = **new** Admin();  // admin.setUserName("Jack");  // admin.setPwd("999");    // b. BeanUtils组件实现对象属性的拷贝  BeanUtils.*copyProperty*(admin, "userName", "jack");  BeanUtils.*setProperty*(admin, "age", 18);    // 总结1： 对于基本数据类型，会自动进行类型转换!      // c. 对象的拷贝  Admin newAdmin = **new** Admin();  BeanUtils.*copyProperties*(newAdmin, admin);    // d. map数据，拷贝到对象中  Admin adminMap = **new** Admin();  Map<String,Object> map = **new** HashMap<String,Object>();  map.put("userName", "Jerry");  map.put("age", 29);  // 注意：map中的key要与javabean的属性名称一致  BeanUtils.*populate*(adminMap, map);    // 测试  System.*out*.println(adminMap.getUserName());  System.*out*.println(adminMap.getAge());  }      //2. 自定义日期类型转换器  @Test  **public** **void** test2() **throws** Exception {  // 模拟表单数据  String name = "jack";  String age = "20";  String birth = " ";    // 对象  Admin admin = **new** Admin();    // 注册日期类型转换器：1， 自定义的方式  ConvertUtils.*register*(**new** Converter() {  // 转换的内部实现方法，需要重写  @Override  **public** Object convert(Class type, Object value) {    // 判断  **if** (type != Date.**class**) {  **return** **null**;  }  **if** (value == **null** || "".equals(value.toString().trim())) {  **return** **null**;  }      **try** {  // 字符串转换为日期  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");  **return** sdf.parse(value.toString());  } **catch** (ParseException e) {  **throw** **new** RuntimeException(e);  }  }  },Date.**class**);        // 把表单提交的数据，封装到对象中  BeanUtils.*copyProperty*(admin, "userName", name);  BeanUtils.*copyProperty*(admin, "age", age);  BeanUtils.*copyProperty*(admin, "birth", birth);    //------ 测试------  System.*out*.println(admin);  }    //2. 使用提供的日期类型转换器工具类  @Test  **public** **void** test3() **throws** Exception {  // 模拟表单数据  String name = "jack";  String age = "20";  String birth = **null**;    // 对象  Admin admin = **new** Admin();    // 注册日期类型转换器：2， 使用组件提供的转换器工具类  ConvertUtils.*register*(**new** DateLocaleConverter(), Date.**class**);    // 把表单提交的数据，封装到对象中  BeanUtils.*copyProperty*(admin, "userName", name);  BeanUtils.*copyProperty*(admin, "age", age);  BeanUtils.*copyProperty*(admin, "birth", birth);    //------ 测试------  System.*out*.println(admin);  }  } |

## 应用

|  |
| --- |
| **public** **class** WebUtils {    @Deprecated  **public** **static** <T> T ~~copyToBean\_old~~(HttpServletRequest request, Class<T> clazz) {  **try** {  // 创建对象  T t = clazz.newInstance();    // 获取所有的表单元素的名称  Enumeration<String> enums = request.getParameterNames();  // 遍历  **while** (enums.hasMoreElements()) {  // 获取表单元素的名称:<input type="password" name="pwd"/>  String name = enums.nextElement(); // pwd  // 获取名称对应的值  String value = request.getParameter(name);  // 把指定属性名称对应的值进行拷贝  BeanUtils.*copyProperty*(t, name, value);  }    **return** t;  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }    /\*\*  \* 处理请求数据的封装  \*/  **public** **static** <T> T copyToBean(HttpServletRequest request, Class<T> clazz) {  **try** {  // （注册日期类型转换器）  // 创建对象  T t = clazz.newInstance();  BeanUtils.*populate*(t, request.getParameterMap());  **return** t;  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  } |

# 元数据

* 在jdbc中获取数据库的定义，例如：数据库、表、列的定义信息。就用到元数据。
* 在jdbc中可以使用： 数据库元数据、参数元数据、结果集元数据
* (元数据定义相关api， 都以MetaData结尾)

元数据： 定义基本数据类型

|  |
| --- |
| **public** **class** App {  //1. 数据库元数据  @Test  **public** **void** testDB() **throws** Exception {  // 获取连接  Connection conn = JdbcUtil.*getConnection*();  // 获取数据库元数据  DatabaseMetaData metaData = conn.getMetaData();// alt + shift + L 快速获取方法返回值    System.*out*.println(metaData.getUserName());  System.*out*.println(metaData.getURL());  System.*out*.println(metaData.getDatabaseProductName());  }    //2. 参数元数据  @Test  **public** **void** testParams() **throws** Exception {  // 获取连接  Connection conn = JdbcUtil.*getConnection*();  // SQL  String sql = "select \* from dept where deptid=? and deptName=?";  // Object[] values = {"tom","888"};    PreparedStatement pstmt = conn.prepareStatement(sql);  // 参数元数据  ParameterMetaData p\_metaDate = pstmt.getParameterMetaData();  // 获取参数的个数  **int** count = p\_metaDate.getParameterCount();      // 测试  System.*out*.println(count);  }    // 3. 结果集元数据  @Test  **public** **void** testRs() **throws** Exception {  String sql = "select \* from dept ";    // 获取连接  Connection conn = JdbcUtil.*getConnection*();  PreparedStatement pstmt = conn.prepareStatement(sql);  ResultSet rs = pstmt.executeQuery();  // 得到结果集元数据(目标：通过结果集元数据，得到列的名称)  ResultSetMetaData rs\_metaData = rs.getMetaData();    // 迭代每一行结果  **while** (rs.next()) {  // 1. 获取列的个数  **int** count = rs\_metaData.getColumnCount();  // 2. 遍历，获取每一列的列的名称  **for** (**int** i=0; i<count; i++) {  // 得到列的名称  String columnName = rs\_metaData.getColumnName(i + 1);  // 获取每一行的每一列的值  Object columnValue = rs.getObject(columnName);  // 测试  System.*out*.print(columnName + "=" + columnValue + ",");  }  System.*out*.println();  }    }        } |

# Dao操作的抽取, BaseDao

Dao操作通用的步骤：

1. 写SQL语句
2. 获取连接
3. 创建stmt
4. 执行sql
   1. 更新
   2. 查询
5. 关闭/异常

通过的dao,

1. 更新

String sql = “select \* from admin”;

String sql = “select \* from admin where id=? And pwd =?”;

public void update(String sql, Object[] paramValues);

1. 查询

String sql = “select \* from admin”;

String sql = “select \* from admin where id=? And pwd =?”;

// 传入的什么类型的对象，就封装为什么类型

// 要求： 列的名称，要与指定类型的对象的属性名称一样

Public List<T> query (String sql , Object[] paramValues , Class<T> clazz);

T t; // 对象赋值

|  |
| --- |
| /\*\*  \* 通用的dao，自己写的所有的dao都继承此类;  \* 此类定义了2个通用的方法：  \* 1. 更新  \* 2. 查询  \* **@author** Jie.Yuan  \*  \*/  **public** **class** BaseDao {    // 初始化参数  **private** Connection con;  **private** PreparedStatement pstmt;  **private** ResultSet rs;  /\*\*  \* 更新的通用方法  \* **@param** sql 更新的sql语句(update/insert/delete)  \* **@param** paramsValue sql语句中占位符对应的值(如果没有占位符，传入null)  \*/  **public** **void** update(String sql,Object[] paramsValue){    **try** {  // 获取连接  con = JdbcUtil.*getConnection*();  // 创建执行命令的stmt对象  pstmt = con.prepareStatement(sql);  // 参数元数据： 得到占位符参数的个数  **int** count = pstmt.getParameterMetaData().getParameterCount();    // 设置占位符参数的值  **if** (paramsValue != **null** && paramsValue.length > 0) {  // 循环给参数赋值  **for**(**int** i=0;i<count;i++) {  pstmt.setObject(i+1, paramsValue[i]);  }  }  // 执行更新  pstmt.executeUpdate();    } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  JdbcUtil.*closeAll*(con, pstmt, **null**);  }  }    /\*\*  \* 查询的通用方法  \* **@param** sql  \* **@param** paramsValue  \*/  **public** <T> List<T> query(String sql, Object[] paramsValue,Class<T> clazz){    **try** {  // 返回的集合  List<T> list = **new** ArrayList<T>();  // 对象  T t = **null**;    // 1. 获取连接  con = JdbcUtil.*getConnection*();  // 2. 创建stmt对象  pstmt = con.prepareStatement(sql);  // 3. 获取占位符参数的个数， 并设置每个参数的值  **int** count = pstmt.getParameterMetaData().getParameterCount();  **if** (paramsValue != **null** && paramsValue.length > 0) {  **for** (**int** i=0; i<paramsValue.length; i++) {  pstmt.setObject(i+1, paramsValue[i]);  }  }  // 4. 执行查询  rs = pstmt.executeQuery();  // 5. 获取结果集元数据  ResultSetMetaData rsmd = rs.getMetaData();  // ---> 获取列的个数  **int** columnCount = rsmd.getColumnCount();    // 6. 遍历rs  **while** (rs.next()) {  // 要封装的对象  t = clazz.newInstance();    // 7. 遍历每一行的每一列, 封装数据  **for** (**int** i=0; i<columnCount; i++) {  // 获取每一列的列名称  String columnName = rsmd.getColumnName(i + 1);  // 获取每一列的列名称, 对应的值  Object value = rs.getObject(columnName);  // 封装： 设置到t对象的属性中 【BeanUtils组件】  BeanUtils.*copyProperty*(t, columnName, value);  }    // 把封装完毕的对象，添加到list集合中  list.add(t);  }    **return** list;  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  JdbcUtil.*closeAll*(con, pstmt, rs);  }  }  } |

|  |
| --- |
| **public** **class** AdminDao **extends** BaseDao {  // 删除  **public** **void** delete(**int** id) {  String sql = "delete from admin where id=?";  Object[] paramsValue = {id};  **super**.update(sql, paramsValue);  }  // 插入  **public** **void** save(Admin admin) {  String sql = "insert into admin (userName,pwd) values (?,?)";  Object[] paramsValue = {admin.getUserName(),admin.getPwd()};  **super**.update(sql, paramsValue);  }    // 查询全部  **public** List<Admin> getAll(){  String sql = "select \* from admin";  List<Admin> list = **super**.query(sql, **null**, Admin.**class**);  **return** list;  }    // 根据条件查询(主键)  **public** Admin findById(**int** id){  String sql = "select \* from admin where id=?";  List<Admin> list = **super**.query(sql, **new** Object[]{id}, Admin.**class**);  **return** (list!=**null**&&list.size()>0) ? list.get(0) : **null**;  }    } |

# 4. DbUtils组件

* commons-dbutils 是 Apache 组织提供的一个开源 JDBC工具类库，它是对JDBC的简单封装，学习成本极低，并且使用dbutils能极大简化jdbc编码的工作量，同时也不会影响程序的性能。因此dbutils成为很多不喜欢hibernate的公司的首选。

DbUtils组件，

1. 简化jdbc操作
2. 下载组件，引入jar文件 : commons-dbutils-1.6.jar

## 实例

|-- DbUtils 关闭资源、加载驱动

|-- QueryRunner 组件的核心工具类：定义了所有的与数据库操作的方法(查询、更新)

Int update(Connection conn, String sql, Object param); 执行更新带一个占位符的sql

Int update(Connection conn, String sql, Object… param); 执行更新带多个占位符的sql

Int[] batch(Connection conn, String sql, Object[][] params) 批处理

T query(Connection conn ,String sql, ResultSetHandler<T> rsh, Object... params) 查询方法

Int update( String sql, Object param);

Int update( String sql, Object… param);

Int[] batch( String sql, Object[][] params)

注意： 如果调用DbUtils组件的操作数据库方法，没有传入连接对象，那么在实例化QueryRunner对象的时候需要传入数据源对象： QueryRunner qr = new QueryRunner(ds);

**DbUtils提供的封装结果的一些对象：**

1. BeanHandler: 查询返回单个对象
2. BeanListHandler: 查询返回list集合，集合元素是指定的对象

3) ArrayHandler, 查询返回结果记录的第一行，封装对对象数组, 即返回：Object[]

4) ArrayListHandler, 把查询的每一行都封装为对象数组，再添加到list集合中

5) ScalarHandler 查询返回结果记录的第一行的第一列 (在聚合函数统计的时候用)

6) MapHandler 查询返回结果的第一条记录封装为map

## 项目优化

对之前的注册功能，优化，引入dbutils组件！

要求：

BaseDao.java

查询、更新的通用的方法！

其他的dao就继承baseDao即可！

练习：

1. DbUtils组件实现更新的测试
2. DbUtils组件实现查询的测试
3. 把项目改为用DbUtils组件实现操作数据库
4. beanUtils组件
5. 元数据、BaseDao编码