# Jdbc

## JdbcUtil

|  |
| --- |
| public class JdbcUtil {   private static final String *url* = "jdbc:mysql://localhost:3306/3g?user=root&password=211314&useUnicode=true&characterEncoding=UTF8";  private static String *driverName* = "com.mysql.jdbc.Driver";   static {  try {  Class.*forName*(*driverName*);  } catch (ClassNotFoundException e) {  e.printStackTrace();  }  }   //获取链接  public static Connection getConnection() throws SQLException {  return DriverManager.*getConnection*(*url*);  }   //释放资源方法  public static void releas(ResultSet rs, Statement st, Connection conn){  try {  if (rs != null) {  rs.close();   }  } catch (SQLException e) {  e.printStackTrace();  }finally{  try {  if(st!=null){  st.close();  }  } catch (SQLException e) {  e.printStackTrace();  }finally{  if(conn!=null){  try {  conn.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  }  }  } } |

## Jdbc-insert

|  |
| --- |
| int i = 0;  String sql = "insert into student (Name,Sex,Age) values(?,?,?)";  Connection conn = null;  PreparedStatement pstmt;  try {  conn=JdbcUtil.getConnection();  pstmt = (PreparedStatement) conn.prepareStatement(sql);  pstmt.setString(1, student.getName());  pstmt.setString(2, student.getSex());  pstmt.setString(3, student.getAge());  i = pstmt.executeUpdate();  JdbcUtil.release(null, pstmt,conn);  } catch (SQLException e) {  e.printStackTrace();  } |

## Jdbc-update

|  |
| --- |
| int i = 0;  String sql = "update students set Age='" + student.getAge() + "' where Name='" + student.getName() + "'";  Connection conn = null;  PreparedStatement pstmt;  try {  conn=JdbcUtil.getConnection();  pstmt = (PreparedStatement) conn.prepareStatement(sql);  i = pstmt.executeUpdate();  jdbcUtil.release(null, pstmt,conn);  } catch (SQLException e) {  e.printStackTrace();  } |

## Jdbc-delete

|  |
| --- |
| int i = 0;  String sql = " delete from students where Name='" + name + "'";  Connection conn = null;  PreparedStatement pstmt;  try {  conn=JdbcUtil.getConnection();  pstmt = (PreparedStatement) conn.prepareStatement(sql);  i = pstmt.executeUpdate();  jdbcUtil.release(null, pstmt,conn);  } catch (SQLException e) {  e.printStackTrace();  } |

## Jdbc-query

|  |
| --- |
| String sql = " select \* from students";  Connection conn = null;  PreparedStatement pstmt;  try {  conn=JdbcUtil.getConnection();  pstmt = (PreparedStatement) conn.prepareStatement(sql);  ResultSet rs = pstmt. executeQuery();      while (rs.next()) {  int id= rs.getInt("id");  String name =rs.getString("name");  String pass=rs.getString("password");  String email = rs.getString("email");  Date birthday =rs.getDate("birthday");  float money= rs.getFloat("money");    }  //行数  **int** rowCount =rs.getRow ();  //列信息  ResultSetMetaData= rs.getMetaData();  //获取列数  int colCount = rs.getMetaData().getColumnCount();  //获取列名  **for**(**int** i=0;i< colCount;i++)  String name=rsmd.getColumnName(i+1);  }  jdbcUtil.release(null, pstmt,conn);  } catch (SQLException e) {  e.printStackTrace();  } |

## preparedStatement和Statement

PreparedStatement尽最大可能提高性能. 因为预编译。

PreparedStatement极大地提高了安全性.

# spring -jdbc

## JdbcDaoSupport

|  |
| --- |
| public abstract class JdbcDaoSupport extends DaoSupport {   private JdbcTemplate jdbcTemplate;    */\*\*  \* Set the JDBC DataSource to be used by this DAO.  \*/* public final void setDataSource(DataSource dataSource) {  if (this.jdbcTemplate == null || dataSource != this.jdbcTemplate.getDataSource()) {  this.jdbcTemplate = createJdbcTemplate(dataSource);  initTemplateConfig();  }  }  //...  } |

## JdbcTemplate

### JdbcTemplate-query

|  |
| --- |
| public <T> T query(  PreparedStatementCreator psc, final PreparedStatementSetter pss, final ResultSetExtractor<T> rse)  throws DataAccessException {  Assert.notNull(rse, "ResultSetExtractor must not be null");  logger.debug("Executing prepared SQL query");  return execute(psc, new PreparedStatementCallback<T>() {  public T doInPreparedStatement(PreparedStatement ps) throws SQLException {  ResultSet rs = null;  try {  if (pss != null) {  pss.setValues(ps);  }  rs = ps.executeQuery();  ResultSet rsToUse = rs;  if (nativeJdbcExtractor != null) {  rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);  }  return rse.extractData(rsToUse);  }  finally {  JdbcUtils.closeResultSet(rs);  if (pss instanceof ParameterDisposer) {  ((ParameterDisposer) pss).cleanupParameters();  }  }  }  });  } |

#### query ---> 通过StatementCallback来查询

|  |
| --- |
| public <T> T execute(StatementCallback<T> action) throws DataAccessException {  Assert.*notNull*(action, "Callback object must not be null");   Connection con = DataSourceUtils.*getConnection*(getDataSource());  Statement stmt = null;  try {  Connection conToUse = con;  if (this.nativeJdbcExtractor != null &&  this.nativeJdbcExtractor.isNativeConnectionNecessaryForNativeStatements()) {  conToUse = this.nativeJdbcExtractor.getNativeConnection(con);  }  stmt = conToUse.createStatement();  applyStatementSettings(stmt);  Statement stmtToUse = stmt;  if (this.nativeJdbcExtractor != null) {  stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);  }  T result = action.doInStatement(stmtToUse);  handleWarnings(stmt);  return result;  }  catch (SQLException ex) {  // Release Connection early, to avoid potential connection pool deadlock  // in the case when the exception translator hasn't been initialized yet.  JdbcUtils.*closeStatement*(stmt);  stmt = null;  DataSourceUtils.*releaseConnection*(con, getDataSource());  con = null;  throw getExceptionTranslator().translate("StatementCallback", *getSql*(action), ex);  }  finally {  JdbcUtils.*closeStatement*(stmt);  DataSourceUtils.*releaseConnection*(con, getDataSource());  } } |

#### query ---> 通过ResultSetExtractor来查询

public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws DataAccessException {  
 Assert.*notNull*(sql, "SQL must not be null");  
 Assert.*notNull*(rse, "ResultSetExtractor must not be null");  
 if (logger.isDebugEnabled()) {  
 logger.debug("Executing SQL query [" + sql + "]");  
 }  
 class QueryStatementCallback implements StatementCallback<T>, SqlProvider {  
 public T doInStatement(Statement stmt) throws SQLException {  
 ResultSet rs = null;  
 try {  
 rs = stmt.executeQuery(sql);  
 ResultSet rsToUse = rs;  
 if (nativeJdbcExtractor != null) {  
 rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);  
 }  
 return rse.extractData(rsToUse);  
 }  
 finally {  
 JdbcUtils.*closeResultSet*(rs);  
 }  
 }  
 public String getSql() {  
 return sql;  
 }  
 }  
 return execute(new QueryStatementCallback());  
}

通过ResultSetExtractor来查询 其实通过StatementCallback来查询

#### query ---> 通过RowMapper来查询

|  |
| --- |
| public <T> List<T> query(String sql, RowMapper<T> rowMapper) throws DataAccessException {  return query(sql, new RowMapperResultSetExtractor<T>(rowMapper)); } |

通过RowMapper查询，其实就是通过ResultSetExtractor来查询

##### 常见的rowMapper

|  |
| --- |
| //自定义RowMapper  public <T> T queryForObject(String sql, RowMapper<T> rowMapper) throws DataAccessException {  List<T> results = query(sql, rowMapper);  return DataAccessUtils.*requiredSingleResult*(results); }  public <T> List<T> query(String sql, RowMapper<T> rowMapper) throws DataAccessException {  return query(sql, new RowMapperResultSetExtractor<T>(rowMapper)); }  // 单列RowMapper  public <T> T queryForObject(String sql, Class<T> requiredType) throws DataAccessException {  return queryForObject(sql, getSingleColumnRowMapper(requiredType)); }  protected <T> RowMapper<T> getSingleColumnRowMapper(Class<T> requiredType) {  return new SingleColumnRowMapper<T>(requiredType); }  public <T> List<T> queryForList(String sql, Class<T> elementType) throws DataAccessException {  return query(sql, getSingleColumnRowMapper(elementType)); }  // Map RowMapper  public Map<String, Object> queryForMap(String sql) throws DataAccessException {  return queryForObject(sql, getColumnMapRowMapper()); }  protected RowMapper<Map<String, Object>> getColumnMapRowMapper() {  return new ColumnMapRowMapper(); }  public List<Map<String, Object>> queryForList(String sql) throws DataAccessException {  return query(sql, getColumnMapRowMapper()); } |

### JdbcTemplate-增删该

#### 无参增删改

|  |
| --- |
| public int update(final String sql) throws DataAccessException {  Assert.*notNull*(sql, "SQL must not be null");  if (logger.isDebugEnabled()) {  logger.debug("Executing SQL update [" + sql + "]");  }  class UpdateStatementCallback implements StatementCallback<Integer>, SqlProvider {  public Integer doInStatement(Statement stmt) throws SQLException {  int rows = stmt.executeUpdate(sql);  if (logger.isDebugEnabled()) {  logger.debug("SQL update affected " + rows + " rows");  }  return rows;  }  public String getSql() {  return sql;  }  }  return execute(new UpdateStatementCallback()); }  public int[] batchUpdate(final String[] sql) throws DataAccessException {  Assert.*notEmpty*(sql, "SQL array must not be empty");  if (logger.isDebugEnabled()) {  logger.debug("Executing SQL batch update of " + sql.length + " statements");  }  class BatchUpdateStatementCallback implements StatementCallback<int[]>, SqlProvider {  private String currSql;  public int[] doInStatement(Statement stmt) throws SQLException, DataAccessException {  int[] rowsAffected = new int[sql.length];  if (JdbcUtils.*supportsBatchUpdates*(stmt.getConnection())) {  for (String sqlStmt : sql) {  this.currSql = sqlStmt;  stmt.addBatch(sqlStmt);  }  rowsAffected = stmt.executeBatch();  }  else {  for (int i = 0; i < sql.length; i++) {  this.currSql = sql[i];  if (!stmt.execute(sql[i])) {  rowsAffected[i] = stmt.getUpdateCount();  }  else {  throw new InvalidDataAccessApiUsageException("Invalid batch SQL statement: " + sql[i]);  }  }  }  return rowsAffected;  }  public String getSql() {  return this.currSql;  }  }  return execute(new BatchUpdateStatementCallback()); }  public <T> T execute(StatementCallback<T> action) throws DataAccessException {  Assert.*notNull*(action, "Callback object must not be null");   Connection con = DataSourceUtils.*getConnection*(getDataSource());  Statement stmt = null;  try {  Connection conToUse = con;  if (this.nativeJdbcExtractor != null &&  this.nativeJdbcExtractor.isNativeConnectionNecessaryForNativeStatements()) {  conToUse = this.nativeJdbcExtractor.getNativeConnection(con);  }  stmt = conToUse.createStatement();  applyStatementSettings(stmt);  Statement stmtToUse = stmt;  if (this.nativeJdbcExtractor != null) {  stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);  }  T result = action.doInStatement(stmtToUse);  handleWarnings(stmt);  return result;  }  catch (SQLException ex) {  // Release Connection early, to avoid potential connection pool deadlock  // in the case when the exception translator hasn't been initialized yet.  JdbcUtils.*closeStatement*(stmt);  stmt = null;  DataSourceUtils.*releaseConnection*(con, getDataSource());  con = null;  throw getExceptionTranslator().translate("StatementCallback", *getSql*(action), ex);  }  finally {  JdbcUtils.*closeStatement*(stmt);  DataSourceUtils.*releaseConnection*(con, getDataSource());  } } |

#### 有参增删改

|  |
| --- |
| public int update(String sql, Object... args) throws DataAccessException {  return update(sql, newArgPreparedStatementSetter(args)); }  public int update(String sql, Object[] args, int[] argTypes) throws DataAccessException {  return update(sql, newArgTypePreparedStatementSetter(args, argTypes)); }  public int update(String sql, PreparedStatementSetter pss) throws DataAccessException {  return update(new SimplePreparedStatementCreator(sql), pss); }  protected int update(final PreparedStatementCreator psc, final PreparedStatementSetter pss)  throws DataAccessException {   logger.debug("Executing prepared SQL update");  return execute(psc, new PreparedStatementCallback<Integer>() {  public Integer doInPreparedStatement(PreparedStatement ps) throws SQLException {  try {  if (pss != null) {  pss.setValues(ps);  }  int rows = ps.executeUpdate();  if (logger.isDebugEnabled()) {  logger.debug("SQL update affected " + rows + " rows");  }  return rows;  }  finally {  if (pss instanceof ParameterDisposer) {  ((ParameterDisposer) pss).cleanupParameters();  }  }  }  }); }  public <T> T execute(PreparedStatementCreator psc, PreparedStatementCallback<T> action)  throws DataAccessException {   Assert.*notNull*(psc, "PreparedStatementCreator must not be null");  Assert.*notNull*(action, "Callback object must not be null");  if (logger.isDebugEnabled()) {  String sql = *getSql*(psc);  logger.debug("Executing prepared SQL statement" + (sql != null ? " [" + sql + "]" : ""));  }   Connection con = DataSourceUtils.*getConnection*(getDataSource());  PreparedStatement ps = null;  try {  Connection conToUse = con;  if (this.nativeJdbcExtractor != null &&  this.nativeJdbcExtractor.isNativeConnectionNecessaryForNativePreparedStatements()) {  conToUse = this.nativeJdbcExtractor.getNativeConnection(con);  }  ps = psc.createPreparedStatement(conToUse);  applyStatementSettings(ps);  PreparedStatement psToUse = ps;  if (this.nativeJdbcExtractor != null) {  psToUse = this.nativeJdbcExtractor.getNativePreparedStatement(ps);  }  T result = action.doInPreparedStatement(psToUse);  handleWarnings(ps);  return result;  }  catch (SQLException ex) {  // Release Connection early, to avoid potential connection pool deadlock  // in the case when the exception translator hasn't been initialized yet.  if (psc instanceof ParameterDisposer) {  ((ParameterDisposer) psc).cleanupParameters();  }  String sql = *getSql*(psc);  psc = null;  JdbcUtils.*closeStatement*(ps);  ps = null;  DataSourceUtils.*releaseConnection*(con, getDataSource());  con = null;  throw getExceptionTranslator().translate("PreparedStatementCallback", sql, ex);  }  finally {  if (psc instanceof ParameterDisposer) {  ((ParameterDisposer) psc).cleanupParameters();  }  JdbcUtils.*closeStatement*(ps);  DataSourceUtils.*releaseConnection*(con, getDataSource());  } } |

## StatementCallback

|  |
| --- |
| public interface StatementCallback<T> { T doInStatement(Statement stmt) throws SQLException, DataAccessException;  } |

#### query ---> 通过ResultSetExtractor来查询-->内部类

|  |
| --- |
| public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws DataAccessException {  Assert.*notNull*(sql, "SQL must not be null");  Assert.*notNull*(rse, "ResultSetExtractor must not be null");  if (logger.isDebugEnabled()) {  logger.debug("Executing SQL query [" + sql + "]");  }  class QueryStatementCallback implements StatementCallback<T>, SqlProvider {  public T doInStatement(Statement stmt) throws SQLException {  ResultSet rs = null;  try {  rs = stmt.executeQuery(sql);  ResultSet rsToUse = rs;  if (nativeJdbcExtractor != null) {  rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);  }  return rse.extractData(rsToUse);  }  finally {  JdbcUtils.*closeResultSet*(rs);  }  }  public String getSql() {  return sql;  }  }  return execute(new QueryStatementCallback()); } |

## ResultSetExtractor

|  |
| --- |
| public interface ResultSetExtractor<T> { T extractData(ResultSet rs) throws SQLException, DataAccessException;  } |

### RowMapperResultSetExtractor

|  |
| --- |
| public class RowMapperResultSetExtractor<T> implements ResultSetExtractor<List<T>> {   private final RowMapper<T> rowMapper;   private final int rowsExpected;  public RowMapperResultSetExtractor(RowMapper<T> rowMapper) {  this(rowMapper, 0);  } public RowMapperResultSetExtractor(RowMapper<T> rowMapper, int rowsExpected) {  Assert.*notNull*(rowMapper, "RowMapper is required");  this.rowMapper = rowMapper;  this.rowsExpected = rowsExpected;  }    public List<T> extractData(ResultSet rs) throws SQLException {  List<T> results = (this.rowsExpected > 0 ? new ArrayList<T>(this.rowsExpected) : new ArrayList<T>());  int rowNum = 0;  while (rs.next()) {  results.add(this.rowMapper.mapRow(rs, rowNum++));  }  return results;  }  } |

## RowMapper

|  |
| --- |
| public interface RowMapper<T> {   T mapRow(ResultSet rs, int rowNum) throws SQLException;  } |

### ColumnMapRowMapper

|  |
| --- |
| public class ColumnMapRowMapper implements RowMapper<Map<String, Object>> {   public Map<String, Object> mapRow(ResultSet rs, int rowNum) throws SQLException {  ResultSetMetaData rsmd = rs.getMetaData();  int columnCount = rsmd.getColumnCount();  Map<String, Object> mapOfColValues = createColumnMap(columnCount);  for (int i = 1; i <= columnCount; i++) {  String key = getColumnKey(JdbcUtils.*lookupColumnName*(rsmd, i));  Object obj = getColumnValue(rs, i);  mapOfColValues.put(key, obj);  }  return mapOfColValues;  }   protected Map<String, Object> createColumnMap(int columnCount) {  return new LinkedCaseInsensitiveMap<Object>(columnCount);  }   protected String getColumnKey(String columnName) {  return columnName;  }  protected Object getColumnValue(ResultSet rs, int index) throws SQLException {  return JdbcUtils.*getResultSetValue*(rs, index);  }  } |

### SingleColumnRowMapper

|  |
| --- |
| public class SingleColumnRowMapper<T> implements RowMapper<T> {   private Class<T> requiredType;    public SingleColumnRowMapper() {  }   public SingleColumnRowMapper(Class<T> requiredType) {  this.requiredType = requiredType;  }   public void setRequiredType(Class<T> requiredType) {  this.requiredType = requiredType;  }    @SuppressWarnings("unchecked")  public T mapRow(ResultSet rs, int rowNum) throws SQLException {  // Validate column count.  ResultSetMetaData rsmd = rs.getMetaData();  int nrOfColumns = rsmd.getColumnCount();  if (nrOfColumns != 1) {  throw new IncorrectResultSetColumnCountException(1, nrOfColumns);  }   // Extract column value from JDBC ResultSet.  Object result = getColumnValue(rs, 1, this.requiredType);  if (result != null && this.requiredType != null && !this.requiredType.isInstance(result)) {  // Extracted value does not match already: try to convert it.  try {  return (T) convertValueToRequiredType(result, this.requiredType);  }  catch (IllegalArgumentException ex) {  throw new TypeMismatchDataAccessException(  "Type mismatch affecting row number " + rowNum + " and column type '" +  rsmd.getColumnTypeName(1) + "': " + ex.getMessage());  }  }  return (T) result;  }   protected Object getColumnValue(ResultSet rs, int index, Class requiredType) throws SQLException {  if (requiredType != null) {  return JdbcUtils.*getResultSetValue*(rs, index, requiredType);  }  else {  // No required type specified -> perform default extraction.  return getColumnValue(rs, index);  }  } protected Object getColumnValue(ResultSet rs, int index) throws SQLException {  return JdbcUtils.*getResultSetValue*(rs, index);  }  @SuppressWarnings("unchecked")  protected Object convertValueToRequiredType(Object value, Class requiredType) {  if (String.class.equals(requiredType)) {  return value.toString();  }  else if (Number.class.isAssignableFrom(requiredType)) {  if (value instanceof Number) {  // Convert original Number to target Number class.  return NumberUtils.*convertNumberToTargetClass*(((Number) value), requiredType);  }  else {  // Convert stringified value to target Number class.  return NumberUtils.*parseNumber*(value.toString(), requiredType);  }  }  else {  throw new IllegalArgumentException(  "Value [" + value + "] is of type [" + value.getClass().getName() +  "] and cannot be converted to required type [" + requiredType.getName() + "]");  }  }  } |

### UserRowMapper（程序员自定义）

|  |
| --- |
| Public class UserRowMapper implements RowMapper <User>{  public User mapRow(ResultSet rs, int rowNum)  throws SQLException {  User user = new User();  user.setId(rs.getString("id"));  user.setMobile(rs.getString("mobile"));  user.setEmail(rs.getString("email"));  user.setCjtime(rs.getDate("cjtime"));  user.setXgtime(rs.getDate("xgtime"));  user.setState(rs.getString("state"));  return user;  }  } |

## PreparedStatementCreator

|  |
| --- |
| public interface PreparedStatementCreator {  PreparedStatement createPreparedStatement(Connection con) throws SQLException; } |

### SimplePreparedStatementCreator

|  |
| --- |
| private static class SimplePreparedStatementCreator implements PreparedStatementCreator, SqlProvider {   private final String sql;   public SimplePreparedStatementCreator(String sql) {  Assert.*notNull*(sql, "SQL must not be null");  this.sql = sql;  }   public PreparedStatement createPreparedStatement(Connection con) throws SQLException {  return con.prepareStatement(this.sql);  }   public String getSql() {  return this.sql;  } } |

## PreparedStatementSetter

|  |
| --- |
| public interface PreparedStatementSetter { void setValues(PreparedStatement ps) throws SQLException;  } |

### ArgumentTypePreparedStatementSetter

|  |
| --- |
| public class ArgumentTypePreparedStatementSetter implements PreparedStatementSetter, ParameterDisposer {   private final Object[] args;   private final int[] argTypes;    public ArgumentTypePreparedStatementSetter(Object[] args, int[] argTypes) {  if ((args != null && argTypes == null) || (args == null && argTypes != null) ||  (args != null && args.length != argTypes.length)) {  throw new InvalidDataAccessApiUsageException("args and argTypes parameters must match");  }  this.args = args;  this.argTypes = argTypes;  }    public void setValues(PreparedStatement ps) throws SQLException {  int parameterPosition = 1;  if (this.args != null) {  for (int i = 0; i < this.args.length; i++) {  Object arg = this.args[i];  if (arg instanceof Collection && this.argTypes[i] != Types.*ARRAY*) {  Collection entries = (Collection) arg;  for (Object entry : entries) {  if (entry instanceof Object[]) {  Object[] valueArray = ((Object[]) entry);  for (Object argValue : valueArray) {  doSetValue(ps, parameterPosition, this.argTypes[i], argValue);  parameterPosition++;  }  }  else {  doSetValue(ps, parameterPosition, this.argTypes[i], entry);  parameterPosition++;  }  }  }  else {  doSetValue(ps, parameterPosition, this.argTypes[i], arg);  parameterPosition++;  }  }  }  }  protected void doSetValue(PreparedStatement ps, int parameterPosition, int argType, Object argValue)  throws SQLException {   StatementCreatorUtils.*setParameterValue*(ps, parameterPosition, argType, argValue);  }   public void cleanupParameters() {  StatementCreatorUtils.*cleanupParameters*(this.args);  }  } |

## 标题1.1

### 标题1.1.1

#### 标题1.1.1.1

##### 标题1.1.1.1.1