JDBC核心技术

第一章：概述

1.1 数据持久化

1 持久化：

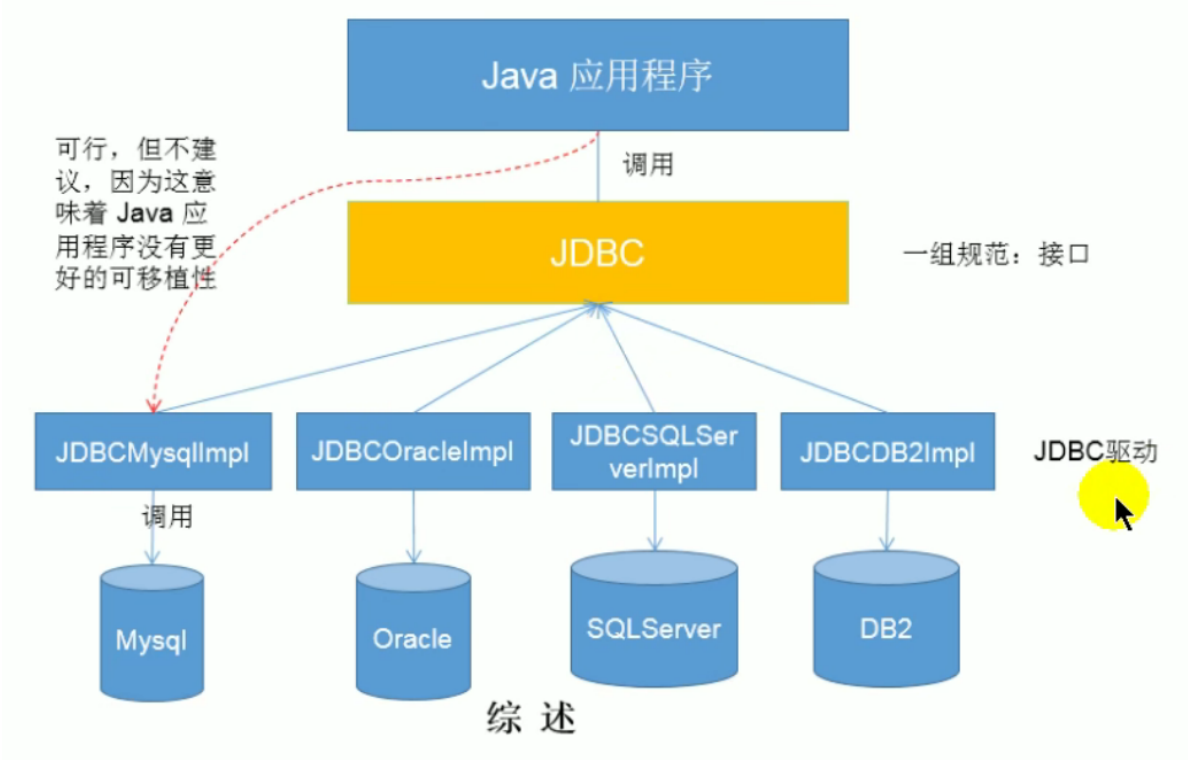
数据保存在可掉电式存储设备中

2 Java中存储技术

- JDBC直接访问数据库
- JDO (Java Data Object) 技术
- 第三方O/R工具，如Hibernate, Mybatis等。封装了JDBC技术

1.2 JDBC

操控数据库的一组API。

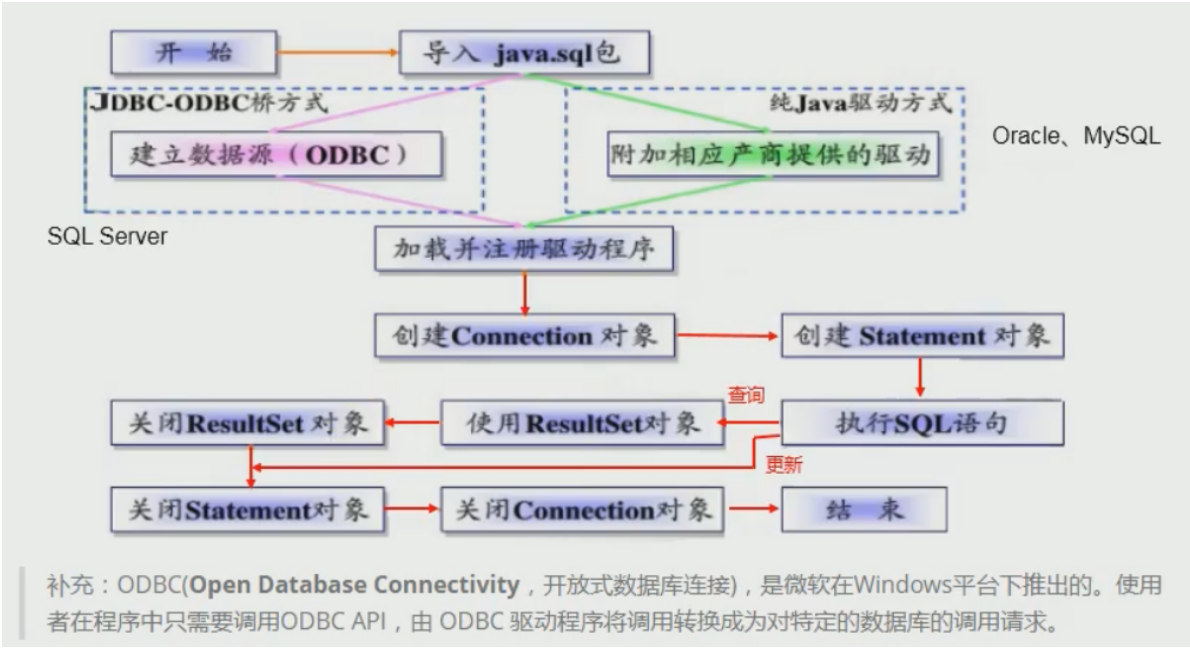


1 JDBC体系结构

- JDBC接口（API）包括两个层次：
  - **面向应用的API**：Java API，抽象接口，供应用程序开发人员使用（连接数据库，执行SQL语句，获得结果）。
  - **面向数据库的API**：Java Driver API，供开发商开发数据库驱动程序用。

> JDBC是sun公司提供一套用于数据库操作的接口，java程序员只需要面向这套接口编程即可。
>
> 不同的数据库厂商，需要针对这套接口，提供不同实现。不同的实现的集合，即为不同数据库的驱动。
>
> ————面向接口编程

2 JDBC编写流程



> 补充：ODBC(**Open Database Connectivity**，开放式数据库连接)，是微软在Windows平台下推出的。使用者在程序中只需要调用ODBC API，由 ODBC 驱动程序将调用转换成为对特定的数据库的调用请求。

第二章：获取数据库的连接

一个数据库连接就是一个Socket连接。

2.1 要素一：Driver接口实现类

java.sql.Driver接口是所有JDBC驱动程序要实现的接口。不同的数据库厂商要提供不同的接口实现。

2.2 要素二：URL

2.2.1 URL

用于标识一个被注册的驱动程序。驱动程序管理器通过这个URL来选择正确的驱动程序，从而建立到数据库的连接。

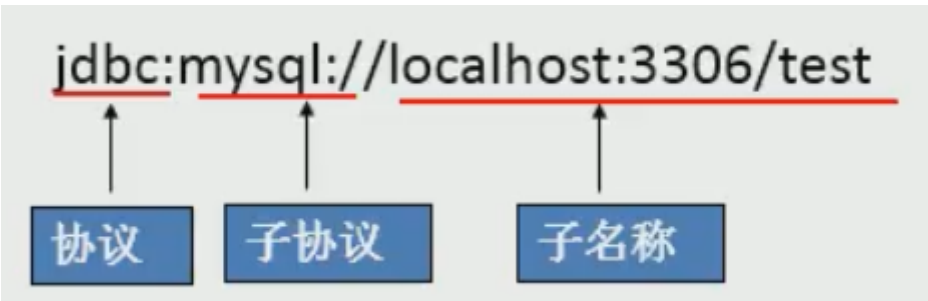
2.2.2 URL组成

类比于http协议下的url，如 http://localhost:80/test/nihao.jpg

JDBC url标准由三部分组成：**jdbc：子协议：子名称**

1. **协议：** JDBC URL中的协议总是jdbc
2. **子协议：**子协议用于标识一个数据库驱动程序
3. **子名称：**一种标识数据库的方法。

举例：



2.3 要素三：Properties

用来保存用户名和密码

```
            //保存用户名和密码
            Properties info = new Properties();
            info.setProperty("user", "root");
            info.setProperty("password", "admin");
```

2.4 连接数据库

2.4.1 方式一：

```
    @Test
    public void test1(){
        Driver driver = null;
        try {
            //获取Driver实现类对象。这里连接mysql数据库
            driver = new com.mysql.jdbc.Driver();

            // jdbc:mysql: 是协议
            // 3306 是mysql端口
            // test 是数据库
            String url = "jdbc:mysql://localhost:3306/test";
            //保存用户名和密码
            Properties info = new Properties();
            info.setProperty("user", "root");
            info.setProperty("password", "admin");

            Connection conn = driver.connect(url, info);
            System.out.println(conn);

        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
    }
```

2.4.2 方式二：

用反射

```
    @Test
    public void test2(){
        Driver driver = null;
        try {
            //1.获取Driver实现类对象：使用反射
            Class clazz = Class.forName("jdbc:mysql://localhost:3306/test");
            driver = (Driver) clazz.newInstance();

            //2.提供要连接的数据库
            String url = "jdbc:mysql://localhost:3306/test";

```

```
            //3.提供用户名和秘密
            Properties info = new Properties();
            info.setProperty("user", "root");
            info.setProperty("password", "admin");

            //4.获取连接
            Connection conn = driver.connect(url, info);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

2.4.3 方式三：

用DriverManager

```
    @Test
    public void test3(){
        try {
            String url = "jdbc:mysql://localhost:3306/test";
            String user = "root";
            String password = "admin";

            //获取连接
            Connection connection = DriverManager.getConnection(url, user,
password);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

2.4.3 方式四：最终版

将数据库连接的4个基本信息放在配置文件中，通过读取配置文件的方式来连接。

```
user=root
password=admin
url=jdbc:mysql://localhost:3306/test
driverClass=com.mysql.jdbc.Driver
```

```
    @Test
    public void test4() {
        try {
            //1.获取配置文件的信息
            //getClassLoader()用系统加载器
            InputStream is =
ConnectionTest.class.getClassLoader().getResourceAsStream("jdbc.properties")
;

            Properties pros = new Properties();
            pros.load(is);

            String user = pros.getProperty("user");
            String password = pros.getProperty("password");
            String url = pros.getProperty("url");
            String driverClass = pros.getProperty("driverClass");
```

```
            //2.加载驱动
            Class.forName(driverClass);

            //3.获取连接
            DriverManager.getConnection(url, user, password);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

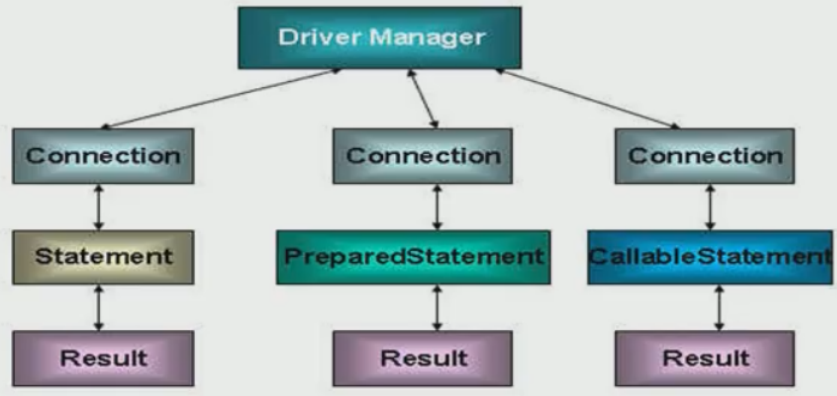
```
    //最后要关闭连接
    conn.close();
```

好处：1. 解耦。实现了数据和代码的分离。2. 如果修改配置文件信息，可以避免程序重新打包。

第三章：PreparedStatement实现CRUD

3.1 调用数据库的三种方式

- 在 java.sql 包中有 3 个接口分别定义了对数据库的调用的不同方式：
  - Statement：用于执行静态 SQL 语句并返回它所生成结果的对象。
  - PrepatedStatement：SQL 语句被预编译并存储在此对象中，可以使用此对象多次高效地执行该语句。
  - CallableStatement：用于执行 SQL 存储过程



3.1 使用Statement的弊端

1. 需要拼写sql语句。字符串拼接很麻烦。
2. 存在sql注入的问题。（用户名密码不对的时候也能操作数据库）
3. 批量插入时效率低。

3.2 PreparedStatement

3.2.1 定义

`PreparedStatement`的好处：

DBserver会预编译语句，以提供性能优化。被DBServer的编译器预编译下来的可执行代码会被缓存，这样相同的语句就不会被编译。

1. `PreparedStatement`是`Statement`的子接口，可以传入带占位符的sql语句，并且提供了补充占位符变量的方法。
2. 可以有效的禁止sql注入。(通过用户输入非法的sql命令)
3. 代码的可读性和可维护性，最大可能的提高性能（批量插入）

3.2.2 Java与SQL对应数据类型转换表

Java类型	SQL类型
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
String	CHAR,VARCHAR,LONGVARCHAR
byte array	BINARY , VAR BINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

3.2.3 代码演示

工具类的封装

ClassLoader.getSystemClassLoader().getResourceAsStream("jdbc.properties");

该方法默认读取的路径是module下src的文件夹目录。

```
/*
* 封装了工具类
*/
public class JDBCUtils {
    /*用于数据库的连接*/
    public static Connection getConnection() throws Exception{
            //1.获取配置文件的信息
            //getClassLoader()用系统加载器
            InputStream is =
ClassLoader.getSystemClassLoader().getResourceAsStream("jdbc.properties");

            Properties pros = new Properties();
            pros.load(is);

            String user = pros.getProperty("user");
            String password = pros.getProperty("password");
            String url = pros.getProperty("url");
            String driverClass = pros.getProperty("driverClass");

```

```
            //2.加载驱动
            Class.forName(driverClass);

            //3.获取连接
            Connection connection = DriverManager.getConnection(url, user,
password);

            return connection;
    }

    /*用于数据库的关闭*/
    public static void closeResource(Connection conn, Statement ps){
        try {
            if(conn != null)
                conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }

        try {
            if(ps != null)
                ps.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }
}
```

增/删/改

```
    /**
    * @Description: UPDATE
    * @Param: [sql, args]: [sql语句，update的数据]
    * @return: null
    * @Author: Finn
    * @Date: 2021/10/17
    */
    public void update(String sql, Object...args){
        Connection connection = null;
        PreparedStatement ps = null;
        try {
            //1.获取数据库连接
            connection = JDBCUtils.getConnection();
            //2.预编译sql语句，返回preparedStatement实例
            ps = connection.prepareStatement(sql);
            //3.填充占位符
            for (int i=0; i<args.length; i++) {
                ps.setObject(i+1, args[i]);//mysql从1开始
            }
            //4.执行sql
            ps.execute();
            //5.资源的关闭
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(connection, ps);
```

```
        }
    }
```

```
注意：
MySQL从1开始计数。
Java数组从0开始计数。
```

查

```
    /**
    * @Description: 针对不同表查询，返回一个对象
    * @Param: [clazz, sql, args] 表，sql语句，数据
    * @return:
    * @Author: Finn
    * @Date: 2021/10/17
    */
    public<T> T getInstance(Class<T> clazz, String sql, Object...args){
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            // 1.获取数据库连接
            conn = JDBCUtils.getConnection();
            // 2.预编译sql语句，得到PreparedStatement对象
            ps = conn.prepareStatement(sql);
            // 3.填充占位符
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);
            }
            // 4.执行executeQuery(),得到结果集：ResultSet
            rs = ps.executeQuery();
            // 5.得到结果集的元数据：ResultSetMetaData
            ResultSetMetaData rsmd = rs.getMetaData();
            // 6.1通过ResultSetMetaData得到columnCount,columnLabel；通过
ResultSet得到列值
            int columnCount = rsmd.getColumnCount();
            if (rs.next()) {
                T t = clazz.newInstance();
                for (int i = 0; i < columnCount; i++) {// 遍历每一个列
                    // 获取列值
                    Object columnVal = rs.getObject(i + 1);
                    // 获取列的别名:列的别名，使用类的属性名充当
                    String columnLabel = rsmd.getColumnLabel(i + 1);
                    // 6.2使用反射，给对象的相应属性赋值
                    Field field = clazz.getDeclaredField(columnLabel);
                    field.setAccessible(true);
                    field.set(t, columnVal);
                }
                return t;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 7.关闭资源
            JDBCUtils.closeResource(conn, ps);
            try {
```

```
                rs.close();
            } catch (SQLException throwables) {
                throwables.printStackTrace();
            }
        }
        return null;
    }
```

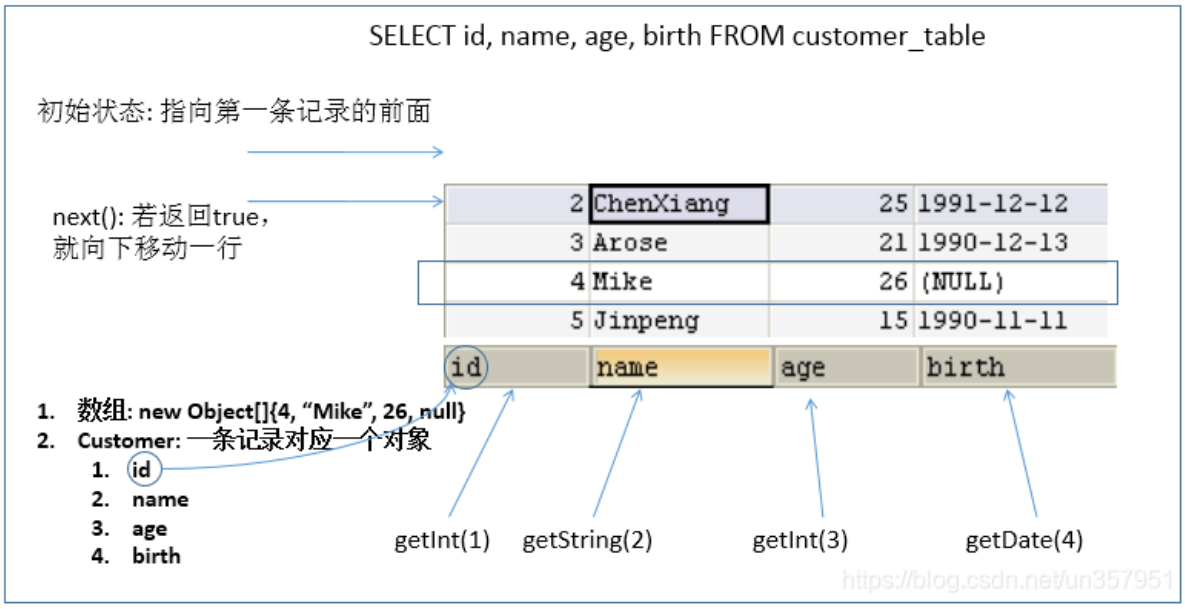
3.3 ResultSet与ResultSetMetaData

3.3.1 ResultSet

`PreparedStatement`实现查询操作的时候，会调用`executeQuery()`，返回结果是`ResultSet`对象。

- `ResultSet`是一个接口，由数据库厂商实现。是以逻辑表格的形式封装了执行数据库操作的结果集。
- `ResultSet`就是一张数据表，有一个指针指向数据表第一条记录的前面。
- 用`boolean next()`方法来操作这个指针。当指针指向一行时，可以通过调用`getXxx(int index)`或`getXxx(int columnName)`获取每一列的值

```
注意：
Java 与数据库交互涉及到的相关 Java API 中的 索引都从 1 开始
```



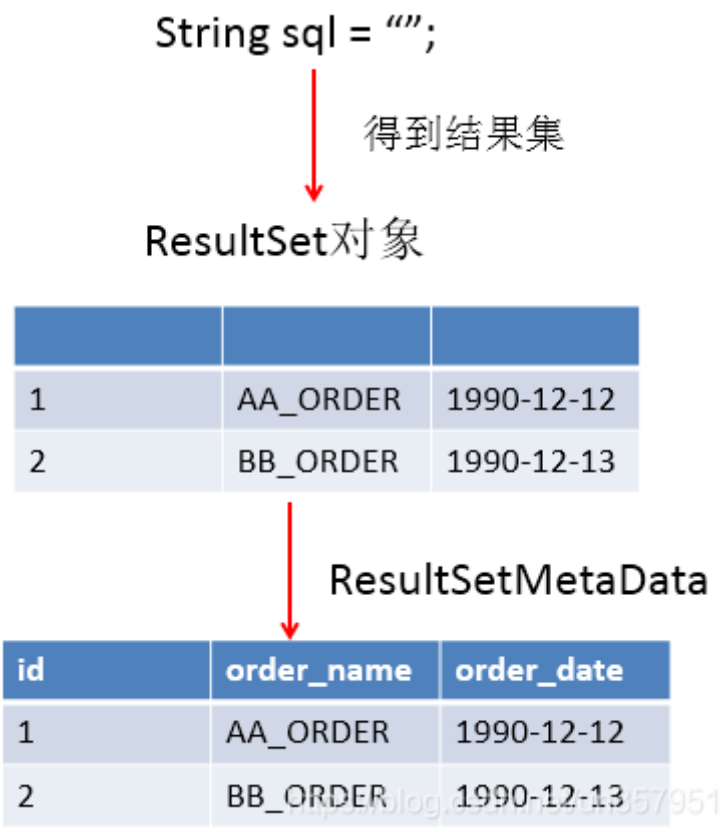
3.3.2 ResultSetMetaData

可用于获取关于`ResultSet`对象中列的类型和属性信息的对象（结果集数据的元数据）

```
ResultSetMetaData meta = rs.getMetaData();
```

1. `getColumnName(int column)`：获取指定列的名称
2. `getColumnLabel(int column)`：获取指定列的别名
3. `getColumnCount()`：返回当前 ResultSet 对象中的列数
4. `getColumnTypeName(int column)`：检索指定列的数据库特定的类型名称
5. `getColumnDisplaySize(int column)`：指示指定列的最大标准宽度，以字符为单位
6. `isNullable(int column)`：指示指定列中的值是否可以为 null
7. `isAutoIncrement(int column)`：指示是否自动为指定列进行编号，这样这些列仍然是只读的

ResultSet和ResultSetMetaData联系



3.4 资源的释放

Statement, Connection, ResultSet 资源都需要`close()`。可以在 finally 中关闭，保证即使其他代码出现异常，资源也一定能被关闭。

1. 数据库连接（Connection）是非常稀有的资源，用完后必须马上释放。
2. 如果 Connection 不能及时正确的关闭将导致系统宕机。
3. Connection 的使用原则是尽量晚创建，尽量早的释放。

3.5 小结

1. 两种思想：
   - 面向编程的思想
   - ORM思想（Object Relational Mapping）
     - 数据表 <——> java类
     - 表中一条记录 <——> java类的一个java对象
     - 表中一条记录的一个字段 <——> java类的一个java对象的一个属性
2. 两种技术
   - `ResultSetMetaData`
     - 获取列数：`getColumnCount()`
     - 获取列的别名：`getColumnLabel()`
   - 反射，创建指定类的对象，获取属性并赋值。

第四章：操作BLOB类型字段

4.1 BLOB

BLOB表示一个不可变、原始数据的类文件对象。可以用文件或二进制的方式读取。

4.2 MySQL中的BLOB类型

类型	大小(单位：字节)
TinyBlob	最大 255
Blob	最大 65K
MediumBlob	最大 16M
LongBlob	最大 4G

4.3 操作BLOB类型字段

需要用`PreparedStatement`来操作BLOB类型的数据。

插入

```
//连接
Connection conn = JDBCUtils.getConnection();

String sql = "insert into
stuinfo(id,name,brith_date,personal_photo)values(?,?,?,?)";
PreparedStatement ps = conn.prepareStatement(sql);

// 填充占位符
ps.setString(1, 16);
ps.setString(2, "James");
ps.setDate(3, new Date(new java.util.Date().getTime()));
// 操作Blob类型的变量
FileInputStream fis = new FileInputStream("xhq.png");
ps.setBlob(4, fis);
//执行
ps.execute();

fis.close();
JDBCUtils.closeResource(conn, ps);
```

修改

```
Connection conn = JDBCUtils.getConnection();
String sql = "update stuinfo set personal_photo = ? where id = ?";
PreparedStatement ps = conn.prepareStatement(sql);

// 填充占位符
// 操作Blob类型的变量
FileInputStream fis = new FileInputStream("coffee.png");
ps.setBlob(1, fis);
ps.setInt(2, 25);

ps.execute();
```

```
12
13 fis.close();
14 JDBCUtils.closeResource(conn, ps);
```

查询

```
1  @Test
2  public void testSelectionForBLOG{
3      String sql = "SELECT id, name, brith_date, personal_photo FROM customer
   WHERE id = ?";
4      Connection conn = null;
5      PreparedStatement ps = null;
6      ResultSet rs = null;
7      try {
8          conn = JDBCUtils.getConnection();
9          ps = conn.prepareStatement(sql);
10         ps.setInt(1, 8);
11         rs = ps.executeQuery();
12
13         if(rs.next()){
14             Integer id = rs.getInt(1);
15             String name = rs.getString(2);
16             Date birthDate = rs.getDate(3);
17             Student stu = new Student(id, name, birthDate);
18             //读取Blob类型的字段
19             Blob blob = rs.getBlob(4);
20             InputStream is = blob.getBinaryStream();
21             OutputStream os = new FileOutputStream("James.jpg");
22             byte [] buffer = new byte[1024];
23             int len = 0;
24             while((len = is.read(buffer)) != -1){
25                 os.write(buffer, 0, len);
26             }
27         }
28     } catch (Exception e){
29         e.printStackTrace();
30     } finally {
31         try{
32             JDBCUtils.closeResource(conn, ps, rs);
33             if(is != null)
34                 is.close();
35             if(os !=  null)
36                 os.close();
37         } catch (Exception e){
38             e.printStackTrace();
39         }
40     }
41 }
```

第五章：批量插入

5.1 批量执行SQL语句

Java批量更新机制

Java批量更新机制允许多条语句一次性提交给数据库，进行批量处理。通常情况下，这样比单独处理更有效率。

JDBC批量处理方法

用`PreparedStatement`实现批量操作数据：

- update、delete本身就具有批量操作的效果。
- 所以，批量操作主要针对insert

用到的方法：

1. `addBatch(String)`：添加需要批量处理的 **SQL 语句**或是**参数**；
    - 多条SQL语句的批量处理；
    - 一条SQL语句的批量传参；
2. `executeBatch()`：执行批量处理语句；
3. `clearBatch()`：清空缓存的数据。

5.2 批量插入

创建goods表：

```
CREATE TABLE goods (
id INT PRIMARY KEY AUTO_INCREMENT,
NAME VARCHAR(25)
);
```

现在想在goods中插入20000条数据。

方式一：Statement（效率低）

```
Connection conn = JDBCUtils.getConnection();
Statement st = conn.createStatement();
for(int i = 1;i <= 20000;i++){
    //每次循环都要创建一个String字符串，占用内存
    String sql = "insert into goods(name) values('name_" + i + "')";
    st.execute(sql);
}
```

方式二：PreparedStatement

```
    /*
    * 使用 addBatch() / executeBatch() / clearBatch() 提高效率
    */
    @Test
    public void testInsert1(){
        Connection conn = null;
        PreparedStatement ps = null;

        String sql = "insert into goods(name) values (?)";

```

```
        try {
            long start = System.currentTimeMillis();

            conn = JDBCUtils.getConnection();
            conn.setAutoCommit(false);//不允许自动提交数据，这样执行操作只是被缓存，而没有真的执行
            ps = conn.prepareStatement(sql);

            for(int i = 1;i <= 20000;i++){
                ps.setString(1, "goods_" + i);
                //1.“攒”sql
                ps.addBatch();
                if(i % 500 == 0){ //每到500的时候insert一次
                    //2.执行
                    ps.executeBatch();
                    //3.清空
                    ps.clearBatch();
                }
            }
            //提交数据
            conn.commit();

            long end = System.currentTimeMillis();
            System.out.println("花费的时间为：" + (end - start));//2521
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, ps);
        }
    }
```

5.3 小结

`PreparedStatement` vs `Statement`：

- 代码的可读性和可维护性。
- **PreparedStatement 能最大可能提高性能：**
  - DBServer会对**预编译**语句提供性能优化。因为预编译语句有可能被重复调用，所以语句在被DBServer的编译器编译后的执行代码被缓存下来，那么下次调用时只要是相同的预编译语句就不需要编译，只要将参数直接传入编译过的语句执行代码中就会得到执行。
  - 在statement语句中,即使是相同操作但因为数据内容不一样,所以整个语句本身不能匹配,没有缓存语句的意义.事实是没有数据库会对普通语句编译后的执行代码缓存。这样每执行一次都要对传入的语句编译一次。
  - (语法检查，语义检查，翻译成二进制命令，缓存)
- PreparedStatement 可以防止 SQL 注入

第六章：数据库事务(☆)

6.1 事务

Transaction

- **Transaction**（事务），指访问并可能更新数据库中各种数据项的**一个程序执行单元**(unit)。一个程序执行单元就是一个或多个DML操作。
- **特性：** 事务是恢复和并发控制的基本单位
- **事务处理（事务操作）：**
  - 保证所有事务都作为一个工作单元来执行，即使出现了故障，都不能改变这种执行方式。
  - 当在一个事务中执行多个操作时：
    - 要么事务中所有的操作都被**提交(commit)**，那么这些修改就永久地保存下来；
    - 要么事务中所有操作被数据库管理系统放弃，整个事务**回滚(rollback)**到最初状态。
- 为确保数据库中数据的**一致性**，数据的操纵应当是离散的成组的逻辑单元：
  - 当全部操作完成时，数据的一致性可以保持;
  - 当一部分操作失败时，整个事务应全部视为错误，所有从起始点以后的操作应**全部回滚(rollback)到开始状态。**

> 数据一旦**提交(commit)**，就不可以再**回滚(callback)**了。
>
> 哪些操作会导致自动提交?
>
> 1. DDL操作执行
>
> i) 可以set autocommit = false 来取消DDL自动提交
>
> 2. DML默认情况下的执行
>
> i) 可以set autocommit = false 来取消DML自动提交
>
> 3. 默认关闭连接时
>
> > 针对数据库连接池:
> >
> > 资源返还到连接池的时候,建议set autocommit = true;再返还

6.2 事务的操作

事务操作流程

```
public void testJDBCTransaction(){
    Connextion conn = null;
    try{
        // 1、获取数据库的连接
        conn = JDBCUtils.getConnection();
        // 2、开启事务
        conn.setAutoCommit(false);
        // 3、进行数据库操作
        // ...
        // 4、如果没有异常，则提交事务
        conn.commit();
    } catch(Exception e) {
        e.printStackTrace();
        // 5、如果有异常，则回滚事务
        try{
            conn.rollback();
        }catch(SQLExcepiton e1){
            e1.printStackTrace();
```

```
        }
    } finally {
        JDBCUtils.close(null,null,conn);
    }
}
```

6.3 事务的ACID属性

ACID	属性	
Atomicity	原子性	原子性是指事务是一个**不可分割**的工作单位，事务中的操作要么都发生，要么都不发生。
Consistency	一致性	事务必须使数据库从一个一致性状态变换到另外一个一致性状态。
Isolation	隔离性	事务内部的操作及使用的数据对并发的其他事务隔离，并发执行的各个事务之间**不能互相干扰。**
Durability	持久性	事务一旦commit，对数据库中数据的改变就是**永久性**的，接下来的其他操作和数据库故障不应该对其有任何影响。

6.4 数据库的并发问题

- 对于同时运行的多个事务，当这些事务访问数据库中相同的数据时，如果没有采取必要的隔离机制，就会导致各种**并发问题**:

  设两个事务 T1, T2：

  a) 脏读: T1读取了已经被 T2 更新但还没有commit的字段。之后，若 T2 callback，T1读取的内容就是临时且无效的。

  b) 不可重复读: T1 读取了一个字段，之后该字段被T2更新并commit。当T1再次读取这一字段时，字段的值就不同了。

  c) 幻读: T1读取一个表，T2 在该表中插入了一些新的行。之后，如果 T1 再次读取同一个表，就会多出几行。

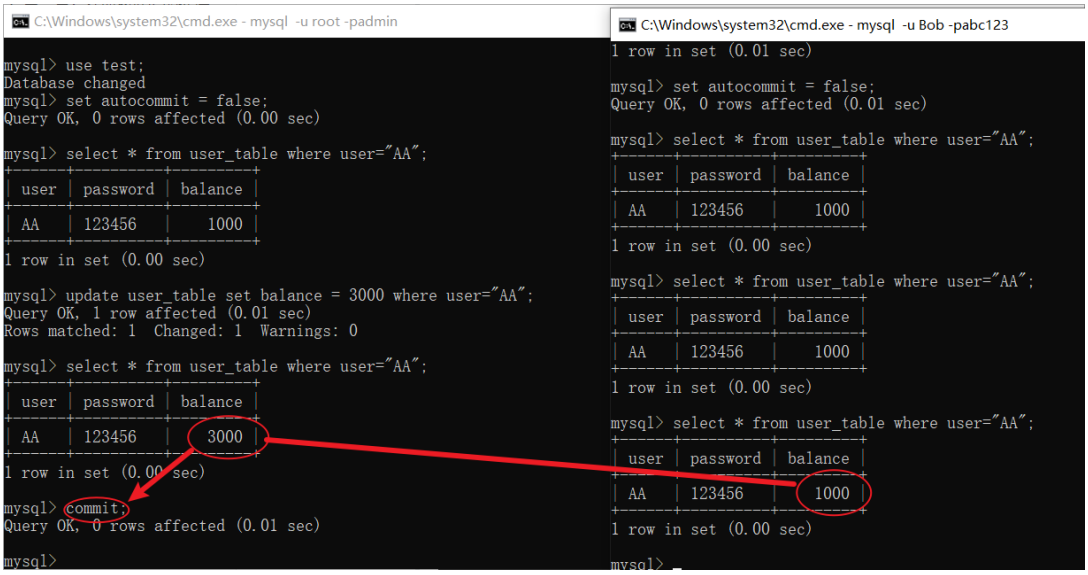
  不可重复读和幻读是可以接受的。脏读是不可以接受的。

- **数据库事务的隔离性**：数据库系统必须具有隔离并发运行各个事务的能力, 使它们不会相互影响, 避免各种并发问题。
- **隔离级别**：一个事务与其他事务隔离的程度称为隔离级别。数据库规定了多种事务隔离级别, 不同隔离级别对应不同的干扰程度, 隔离级别越高, 数据一致性就越好, 但并发性越弱。

6.5 四种隔离级别

隔离级别	描述
READ UNCOMMITTED (读未提交数据)	允许事务读取未被其他事物提交的变更. 脏读, 不可重复读和幻读的问题都会出现
READ COMMITED (读已提交数据)	只允许事务读取已经被其它事务提交的变更. 可以避免脏读, 但不可重复读和幻读问题仍然可能出现
REPEATABLE READ (可重复读)	确保事务可以多次从一个字段中读取相同的值. 在这个事务持续期间, 禁止其他事物对这个字段进行更新. 可以避免脏读和不可重复读, 但幻读的问题仍然存在.
SERIALIZABLE(串行化)	确保事务可以从一个表中读取相同的行. 在这个事务持续期间, 禁止其他事务对该表执行插入, 更新和删除操作. 所有并发问题都可以避免, 但性能十分低下.

- Oracle 支持的 2 种事务隔离级别：**READ COMMITED**, SERIALIZABLE。 Oracle 默认的事务隔离级别为: **READ COMMITED** 。
- Mysql 支持 4 种事务隔离级别。Mysql 默认的事务隔离级别为: **REPEATABLE READ**。

> **演示MySQL的REPEATABLE READ的隔离级别：**
>
> > 左边是root的事务，右边是创建的用户Bob的事务。
> >
> > 可以看到，左边root的事务commit了update操作,使AA的balance变成了3000；但Bob的事务没有commit前查到AA的balance值仍然是1000。
>
> 
>
>
> 所以，只要Bob事务没关闭，他不会读取到别的事务commit的操作。避免了脏读和可重复读的问题。

> **如果隔离等级是READ COMMIT:**
>
> > 那么在root事务update AA的balance为3000，并commit后；就算Bob事务没有关闭，查到AA的balance也是3000了。
> >
> > 这就是不可重复读问题。（我的事务还没结束，却能读到被别的事务修改过的值）

第七章：DAO及相关实现类

第八章：数据库连接池

8.1 JDBC数据库连接池的必要性

在使用开发基于数据库的 web 程序时，传统的模式基本是按以下步骤：

- 在主程序（如 servlet、beans ）中建立数据库连接
- 进行 sql 操作
- 断开数据库连接

这种模式开发，存在的**问题**:

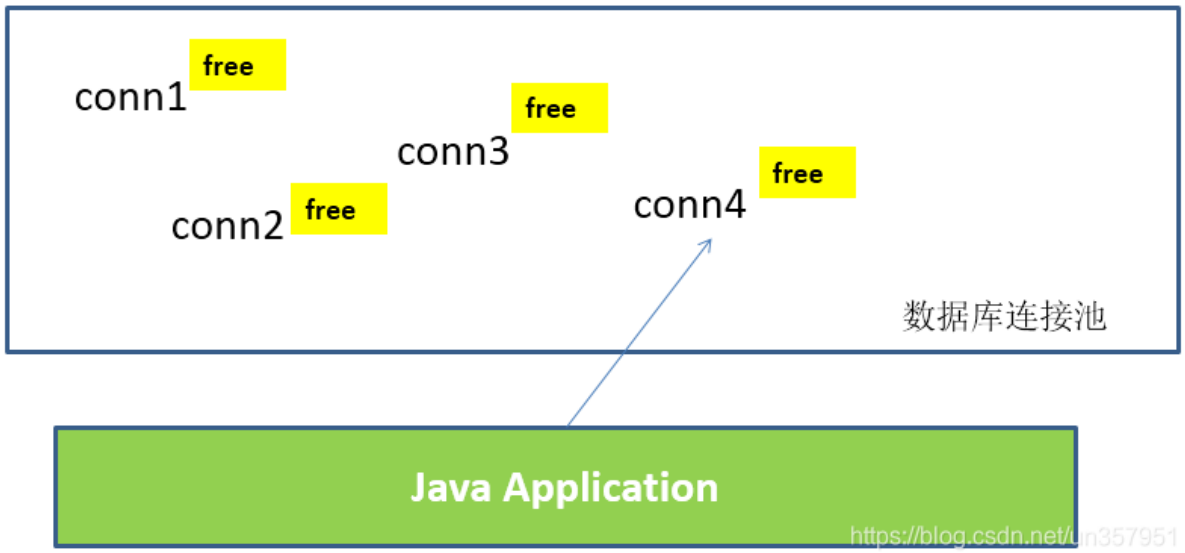
1. 普通的 JDBC 数据库连接使用 `DriverManager` 来获取，每次向数据库建立连接的时候都要将 `Connection` 加载到内存中，再验证用户名和密码(得花费0.05s～1s的时间)。需要数据库连接的时候，就向数据库要求一个，执行完成后再断开连接。这样的方式将会消耗大量的资源和时间。**数据库的连接资源并没有得到很好的重复利用。** 若同时有几百人甚至几千人在线，频繁的进行数据库连接操作将占用很多的系统资源，严重的甚至会造成服务器的崩溃。
2. **对于每一次数据库连接，使用完后都得断开。** 否则，如果程序出现异常而未能关闭，将会导致数据库系统中的内存泄漏，最终将导致重启数据库。
    - 何为 Java 的内存泄漏?
        - 内存有对象，却不能回收的情况
3. **这种开发不能控制被创建的连接对象数**，系统资源会被毫无顾及的分配出去，如连接过多，也可能导致内存泄漏，服务器崩溃。

> 针对上面的问题，提出数据库连接池的技术。

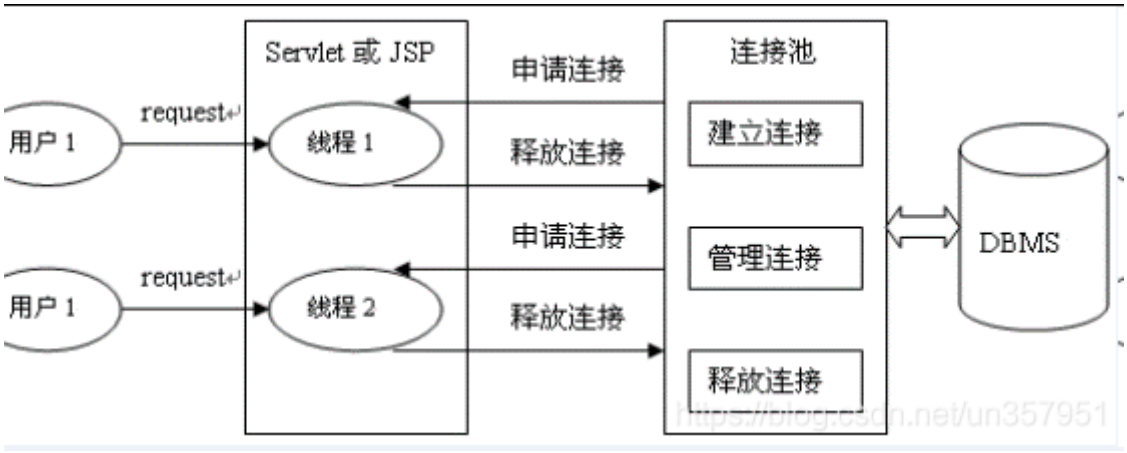
8.2 数据库连接池技术

1. 数据库连接池技术：

- **数据库连接池的基本思想：**就是为数据库连接建立一个pool。预先在pool中放入一定数量的连接，当需要建立数据库连接时，只需从 pool中取出一个，使用完毕之后再放回去。
- 数据库连接池负责分配、管理和释放数据库连接，它**允许应用程序重复使用一个现有的数据库连接，而不是重新建立一个。**
- 数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，这些数据库连接的数量是由**最小数据库连接数来设定**的。无论这些数据库连接是否被使用，连接池都将一直保证至少拥有这么多的连接数量。连接池的**最大数据库连接数量**限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中。



2. 工作原理：



3. 数据库连接池技术的**优点**：

a) 资源重用

由于数据库连接得以重用，避免了频繁创建，释放连接引起的大量性能开销。在减少系统消耗的基础上，另一方面也增加了系统运行环境的平稳性。

b) 更快的系统反应速度

数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于连接池中备用。此时连接的初始化工作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而减少了系统的响应时间。

c) 新的资源分配手段

对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接池的配置，实现某一应用最大可用数据库连接数的限制，避免某一应用独占所有的数据库资源。

d) 统一的连接管理，避免数据库连接泄漏

在较为完善的数据库连接池实现中，可根据预先的占用超时设定，强制回收被占用连接，从而避免了常规数据库连接操作中可能出现的资源泄露。

8.3 多种开源的数据库连接池

- JDBC 的数据库连接池使用`javax.sql.DataSource`来表示，`DataSource`只是一个接口，该接口通常由服务器 (Weblogic, WebSphere, Tomcat) 提供实现，也有一些开源组织提供实现：
  - **DBCP**是 Apache 提供的数据库连接池。Tomcat 服务器自带DBCP数据库连接池。速度相对C3P0较快，但因自身存在BUG，Hibernate3已不再提供支持。
  - **C3P0**是一个开源组织提供的一个数据库连接池，速度相对较慢，稳定性还可以。 Hibernate官方推荐使用
  - **Proxool**是sourceforge下的一个开源项目数据库连接池，有监控连接池状态的功能，稳定性较C3P0差一点
  - **BoneCP**是一个开源组织提供的数据库连接池，速度快
  - **Druid**是阿里提供的数据库连接池，据说是集DBCP 、C3P0 、Proxool优点于一身的数据库连接池，但是速度不确定是否有 BoneCP快
- `DataSource`通常被称为数据源，它包含**连接池**和**连接池管理**两个部分，习惯上也经常把`DataSource`称为连接池
- `DataSource`用来取代`DriverManager`来获取`Connection`，获取速度快，同时可以大幅度提高数据库访问速度。

> 特别注意：
>
> > - 一个应用创建一个数据源：数据源和数据库连接不同，数据源无需创建多个，它是产生数据库连接的工厂，因此整个应用只需要一个数据源即可。

> - 当数据库访问结束后，程序还是像以前一样关闭数据库连接：conn.close(); 但 conn.close() 并没有关闭数据库的物理连接，它仅仅把数据库连接释放，归还给了数据库连接池。

8.4 Druid数据库连接池

```
public class JDBCUtils {
    //Druid连接池
    private static DataSource source;
    static{
        Properties pros = null;
        InputStream is = null;
        Connection conn = null;
        try {
            pros = new Properties();
            is = ClassLoader.getSystemClassLoader().getResourceAsStream("druid.properties");
            pros.load(is);
            source = DruidDataSourceFactory.createDataSource(pros);//ctrl + alt + 鼠标左键
            conn = source.getConnection();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static Connection getConnection() throws SQLException {
        Connection conn = source.getConnection();
        return conn;
    }
}
```

> ctrl+alt +鼠标左键 或 ctrl+alt+b：查看implementation of Interface

druid.properties：

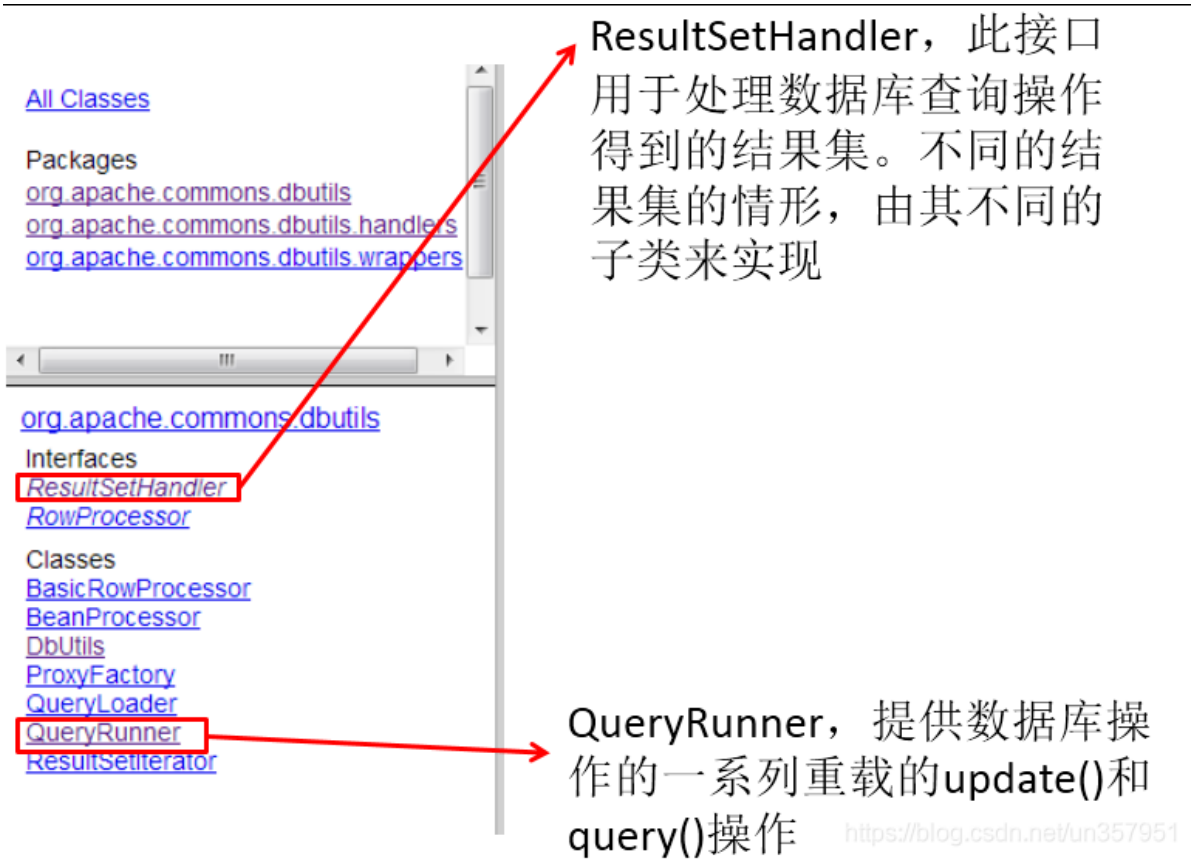
```
url=jdbc:mysql://localhost:3306/test?rewriteBatchedStatements=true
username=root
password=123456
driverClassName=com.mysql.jdbc.Driver

initialSize=10
maxActive=20
maxWait=1000
filters=wall
```

第九章：Apache-DBUtils

9.1 Apache-DBUtils简介

- `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。
- API介绍：
  - org.apache.commons.dbutils.QueryRunner
  - org.apache.commons.dbutils.ResultSetHandler
  - 工具类：org.apache.commons.dbutils.DbUtils
- API包说明：



9.2 QueryRunner类

```
public class QueryRunnerTest {
    //增删改
    @Test
    public void testQuery(){
        Connection conn = null;
        try {
```

```
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection();
            String sql = "select name from stuinfo where id=?";
            BeanListHandler<Student> rsh = new BeanListHandler<>
(Student.class);
            List<Student> students = runner.query(conn, sql, rsh, 6);
            students.forEach(System.out::println);
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn,null);
        }
    }

    //查询
    @Test
    public void testInsert(){
        Connection conn = null;
        try {
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection();
            String sql = "insert into stuinfo(id,name) values(?,?)";
            runner.update(conn,sql,6,"Amy");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

`ResultSetHander`保存了查询到的结果，可以自定义匿名类来实现`ResultSetHander`接口：

```
    @Test
    public void testMyQuery(){
        Connection conn = null;
        try {
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection();
            String sql = "select name from stuinfo where id=?";
            ResultSetHandler<Student> rsh = new ResultSetHandler<Student>()
{
                @Override
                public Student handle(ResultSet resultSet) throws
SQLException {
                    if(resultSet.next()){
                        String name = resultSet.getString("name");
                        return new Student(6, name);
                    }
                    return null;
                }
            };
            Student s = runner.query(conn, sql, rsh, 6);
            System.out.println(s);
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn,null);
        }
```

```
25      }
```

9.3 DBUtils关闭资源

使用dbutils.jar中提供的DbUtils工具类，来实现资源的关闭

```
1         /*
2         * DBUtils关闭资源
3         * */
4         public static void dbUtilsCloseResource(Connection conn, Statement
ps, ResultSet rs){
5                 DbUtils.closeQuietly(conn, ps, rs);
6         }
```