JDBC

JDBC（Java DataBase Connectivity），是 Sun 公司提供的一套 **操作数据库的标准规范。**JDBC 提供一些操作数据的 API，开发者可以在 Java 中使用这些 API 操作数据库，JDBC 相当于 **Java 和数据库之间的一座桥梁**

Sun 公司制定了 JDBC 标准，各大数据库厂商会提供数据库驱动现这个标准，这样 Java 才可以通过 JDBC 来操作实现了这个标准的数据库。各个数据库厂商会将各自的数据库驱动打成 JAR 包对外发布，开发者在使用时需要下载与当前数据库匹配的数据库驱动 JAR 包

- JDBC 规范让 Java 程序和数据库驱动实现了 **松耦合**，使切换不同的数据库变得更加简单

JDBC 是如何实现 Java 程序和 JDBC 驱动的松耦合的

通过 Java 的 **反射机制** 来实现松耦合。所有操作都是通过 JDBC 接口完成的，而驱动只有在通过 Class.forName 反射机制来加载的时候才会出现

四大核心接口

- DriverManager：用于 **注册驱动**，并创建符合该驱动的数据库的连接
- Connection：表示与数据库创建的连接对象，即 **一个连接对应着和数据库服务器建立的一个会话**
- Statement：操作数据库 SQL 语句的对象
- ResultSet：从数据库中查询的结果集

java.util.Date 和 java.sql.Date 区别

- java.util.Date **包含日期和时间**，而 java.sql.Date **只包含日期信息**，而没有具体的时间信息
- 如果想把时间信息存储在数据库里，可以考虑使用 Timestamp 或者 DateTime 字段

CLOB 和 BLOB 数据类型

- CLOB：**字符大对象**，由具有关联代码页的单字节字符组成的字符串
  - 适用于存储面向文本的信息，信息量可能超出常规 varchar 数据类型的限制
- BLOB：**二进制大对象**，由字节组成的二进制字符串，没有关联的代码页
  - 适用于存储图像，语音，图形和其他类型的业务或特定于应用程序的数据。此数据类型可以存储大于 varbinary 的二进制数据

使用 JDBC

JDBC 编程步骤

1. 注册驱动
2. 获取连接 Connection
3. 得到执行 SQL 语句的对象 Statement
4. 执行 SQL 语句，并返回结果
5. 处理结果
6. 关闭 Connection

```java
// 注册驱动
Class.forName("com.mysql.jdbc.Driver");
// 获取连接
String url = "jdbc:mysql://localhost:3306/test";
String username = "root";
String password = "1234";
Connection con = DriverManager.getConnection(url, username, password);
// 得到执行SQL语句的对象Statement
Statement st = con.createStatement();
// 执行SQL语句并返回结果
ResultSet rs = st.executeQuery("select * from user");
// 处理结果
while (rs.next()) {
    System.out.print(rs.getObject("id") + "\t");
    System.out.println(rs.getObject("name"));
}
// 关闭资源
rs.close();
st.close();
con.close();
```

JDK7 和 JDBC4.1 之后的正确关闭资源

Connection、Statement、ResultSet 都继承了 AutoCloseable 接口，因此可以使用 try-with-resources 的方式关闭这些资源

```java
String url = "jdbc:mysql://localhost:3306/test";
String username = "root";
String password = "1234";
String sql = "select * from user";

try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}

try (Connection con = DriverManager.getConnection(url, username, password);
     Statement st = con.createStatement();
     ResultSet rs = st.executeQuery(sql))
    {
    while (rs.next()) {
        System.out.print(rs.getObject("id") + "\t");
        System.out.println(rs.getObject("name"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

使用工具类注册驱动和获取连接

数据库配置文件：db.properties

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/abcd
username=root
password=1234
```

工具类：DBUtil

```
private static String driver;
private static String url;
private static String username;
private static String password;

static {
    ResourceBundle rb = ResourceBundle.getBundle("db");
    driver = rb.getString("driver");
    url = rb.getString("url");
    username = rb.getString("username");
    password = rb.getString("password");
    try {
        Class.forName(driver);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}

public static Connection getConnection() throws SQLException {
    return DriverManager.getConnection(url, username, password);
}
```

测试类

```
String sql = "select * from user";

try (Connection con = DBUtil.getConnection();
     Statement st = con.createStatement();
     ResultSet rs = st.executeQuery(sql))
    {
    while (rs.next()) {
        System.out.print(rs.getObject("id") + "\t");
        System.out.println(rs.getObject("name"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

JDBC 常用接口

DriverManager

主要作用就是创建连接，使用了反射机制注册驱动，不同的数据库，在 forName 中的参数写法不同。通过调用 getConnection 方法，来获取连接

Statement

主要作用是操作 SQL 语句，并返回相应结果的对象

```
// 根据查询语句返回结果集，只能执行select语句
ResultSet executeQuery(String sql)

// 根据执行的DML（insert update delete）语句，返回受影响的行数
int executeUpdate(String sql)

// 可以执行任意SQL语句。返回boolean值，表示是否返回ResultSet结果集
// 仅当执行select语句，且有返回结果时返回true，其它语句都返回false
// 常用于执行不明确的SQL语句，尽量避免使用
boolean execute(String sql)

// 获取自动生成的主键的值
ResultSet getGeneratedKeys();

// 限制数据库从查询返回的行数
void setMaxRows(int max);

// 如果有一个返回100行的查询，将fetchSize设置为10
// 在每次数据库访问时，只会获取10行，通过10次访问以获取所有行
// 如果每行需要大量处理时间并且结果中的行数很大，那么设置最佳fetchSize会很有帮助
void setFetchSize(int rows);
```

增删改查

```
// 增
String insert = "insert into user values (999,'test')";
int flag = st.executeUpdate(insert);
if (flag > 0) {
    System.out.println("success");
}
// 删
String delete = "delete from user where id = 1";
int flag = st.executeUpdate(delete);
if (flag > 0) {
    System.out.println("success");
}
// 改
String delete = "update user set name = 'change' where id = 999";
int flag = st.executeUpdate(update);
if (flag > 0) {
    System.out.println("success");
}
// 查
String select = "select id,name from user";
ResultSet rs = st.executeQuery(select);
```

```
while (rs.next()) {
    System.out.print(rs.getObject("id") + "\t");
    System.out.println(rs.getObject("name"));
}
```

ResultSet

主要用来封装结果集。ResultSet 对象维护了一个游标，指向当前的数据行。开始的时候这个游标指向的是第一行。如果调用了 next() 游标会下移一行，如果没有更多的数据了，next() 会返回 false

```
// 根据列名取值
getObject(String ColomnName);

// 根据序号取值，索引从1开始，可读性不强，不建议使用
getObject(int columnIndex);
```

在做查询操作时，可能会返回多条数据结果，可以定义一个实体类，将数据封装到实体类中

```
String select = "select id,name from user";
ResultSet rs = st.executeQuery(select);
List<User> userList = new ArrayList<>();

while(rs.next()){
    User user = new User();
    u.setId(rs.getInt("id"));
    u.setName(rs.getString("name"));
    userList.add(user);
}
```

- 当生成 ResultSet 的 Statement 对象要关闭、或者重新执行、或者获取下一个 ResultSet 时，ResultSet 对象会自动关闭

PreparedStatement

继承自 Statement，代表的是一个 **预编译** 的 SQL 语句

- 性能比 Statement 高，会把 SQL 预编译
- 可以解决 SQL 注入问题
- 可以进行 **动态查询**
- 执行单一查询，性能非常慢，不建议使用

```
// 在sql语句中，使用?作为占位符来替代要传入的内容
// 通过调用PreparedStatement的setString等方法将要传入的内容作为参数传递过去
String sql = "select * from user where id = ? and name = ?";

try (Connection con = DBUtil.getConnection();
     PreparedStatement ps = con.prepareStatement(sql);
) {
    ps.setInt(1, 1);
    ps.setString(2, "test");

```

```
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            System.out.print(rs.getInt("id") + "\t");
            System.out.println(rs.getString("name"));
        }
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

如何注入 Null 值

可以使用 setNull 方法来把 Null 值绑定到指定的变量上，需要传入参数的索引以及 SQL 字段的类型

批处理

可以一次性为数据库执行大量查询，JDBC 支持通过 Statement 和 PreparedStatement 的 addBatch() 以及 executeBatch() 进行批处理。批处理比一次执行一个语句更快，因为数据库调用的数量较少

```
String add = "insert into user values (?,?)";

try (Connection con = DBUtil.getConnection();
     PreparedStatement ps = con.prepareStatement(add)
) {
    for (int i = 0; i < 200; i++) {
        ps.setInt(1, i + 100);
        ps.setString(2, "test" + (i + 100));

        // 添加到批处理中
        ps.addBatch();
        if (i % 2 == 100) {
            // 执行批处理
            ps.executeBatch();
            // 清空批处理
            // 如果数据量太大，所有数据存入批处理，内存肯定溢出
            ps.clearBatch();
        }
    }
    // 不是所有的%2==100，剩下的再执行一次批处理
    ps.executeBatch();
    // 再清空
    ps.clearBatch();

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

RowSet

继承自 ResultSet，用于存储查询的数据结果，比 ResultSet 更具灵活性。**提供了 JavaBean 的功能**，可以通过 set 和 get 方法来设置和获取属性。RowSet 使用了 JavaBean 的事件驱动模型，可以给注册的组件发送事件通知，如游标的移动，行的增删改，以及 RowSet 内容的修改等

RowSet 对象默认是可滚动，可更新的，因此如果数据库系统不支持 ResultSet 实现类似的功能，可以使用 RowSet 来实现。RowSet 分为两大类

- **连接型 RowSet**：这类对象与数据库进行连接，和 ResultSet 类似。JDBC 接口只提供了一种连接型 RowSet
  - JdbcRowSet
- **离线型RowSet**：这类对象 **不需要和数据库进行连接**，更轻量级，更容易序列化，适用于在网络间传递数据
  - **CachedRowSet**：可以通过他们获取连接，执行查询并读取 ResultSet 的数据到 RowSet 里。可以在离线时对数据进行维护和更新，然后重新连接到数据库里，并回写改动的数据
  - **WebRowSet**：继承自 CachedRowSet，可以读写 XML 文档
  - **JoinRowSet**：继承自 WebRowSet，不用连接数据库就可以执行 SQL 的 join 操作
  - **FilteredRowSet**：继承自 WebRowSet，可以用它来设置过滤规则，这样只有选中的数据才可见

RowSet 和 ResultSet 的区别

RowSet 继承自 ResultSet，因此有 ResultSet 的全部功能，同时添加了些额外的特性。RowSet 最大的好处就是可以离线，这样使得它更轻量级，同时便于在网络间进行传输

事务管理

默认情况下，创建数据库连接时，它会以自动提交模式运行。意味着无论何时执行查询并完成查询，都会自动触发提交。因此，我们触发的每个 SQL 查询都是一个事务，如果我们进行一些增删改操作，则每个 SQL 语句完成后，更改都会保存到数据库中

Connection 对象提供了 `void setAutoCommit(boolean flag)`，可以禁用连接的自动提交功能。应该 **仅在需要时禁用自动提交**，因为除非在连接上调用 commit 方法，否则不会提交事务。可以使用 rollback 方法来回滚事务。它将回滚事务所做的所有更改，并释放此 Connection 对象当前持有的所有数据库锁

Savepoint

有时，事务可以是多个语句的组，如果想要回滚到事务中的特定点。可以使用 setSavepoint 方法在事务中创建检查点，并且可以回滚到该特定检查点

为事务创建的任何保存点都会自动释放，并在提交事务时或在回滚整个事务时变为无效

JDBC 连接隔离级别

可以通过 getTransactionIsolation 方法获取隔离级别信息，并使用 setTransactionIsolation 方法设置它

- TRANSACTION_READ_UNCOMMITTED：读未提交
- TRANSACTION_READ_COMMITTED：读已提交
- TRANSACTION_REPEATABLE_READ：可重复读
- TRANSACTION_SERIALIZABLE：串行化
- TRANSACTION_NONE：不使用隔离级别

DataSource

数据源，跟 DriverManager 相比，功能要更强大。可以用它来创建数据库连接，当然驱动的实现类会实际去完成这个工作。除了能创建连接外，它还提供了如下的特性

- 缓存 PreparedStatement 以便更快的执行
- 可以设置连接超时时间
- 提供日志记录的功能
- ResultSet 大小的最大阈值设置
- 通过 JNDI 的支持，可以为 Servlet 容器提供连接池的功能

JDBC 最佳实践

- 数据库资源是非常昂贵的，用完了应该尽快关闭它
  - 调用 Connection、Statement、ResultSet 等 JDBC 对象的 close 方法
  - 养成在代码中显式关闭掉 ResultSet，Statement，Connection 的习惯，如果你用的是连接池的话，连接用完后会放回池里，但是没有关闭的 ResultSet 和 Statement 会造成资源泄漏
  - 在 finally 块中关闭资源，保证即使出了异常也能正常关闭
- 大量类似的查询应当使用批处理完成
- 尽量使用 PreparedStatement，以避免 SQL 注入，同时还能通过预编译和缓存机制提升执行的效率
- 如果要将大量数据读入到 ResultSet 中，应该合理的设置 fetchSize 以便提升性能
- 你用的数据库可能没有支持所有的隔离级别，用之前先仔细确认下
- 数据库隔离级别越高性能越差，确保数据库连接设置的隔离级别是最优的
- 如果在 Web 应用中创建数据库连接，最好通过 JNDI 使用 JDBC 的数据源，这样可以 **对连接进行重用**
- 如果需要长时间对 ResultSet 进行操作，尽量使用离线的 RowSet，这样可以 **释放数据库连接**

数据库连接池

主要用来 **分配、管理、释放数据库的连接。**数据库连接池首先会创建若干个 Connection 对象并将这些对象放入到池中，当系统需要使用 Connection 对象时，数据库连接池会从池中分配一个事先创建好的 Connection 对象给系统，当系统使用完毕或超时后，数据库连接池会将该 Connection 对象重新放入到池中。这样就减少了创建 Connection 对象所耗费的资源和时间，可以提高对数据库操作的性能

- 数据库的连接的建立和关闭是非常消耗资源的，频繁地打开、关闭连接造成系统性能低下

数据库连接池的作用

- **资源重用**：由于数据库连接得到重用，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，另一方面也增进了系统运行环境的平稳性，减少了内存碎片以及数据库临时进程和线程的数量
- **更快的系统响应速度**：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，从而缩减了系统整体响应时间
- **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的配置，实现数据库连接池技术。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数据库资源
- **统一的连接管理，避免数据库连接泄漏**：在较为完备的数据库连接池实现中，可根据预先的连接占用超时设定，强制收回被占用连接。从而避免了常规数据库连接操作中可能出现的资源泄漏

连接池类

连接池类是对某一数据库所有连接的缓冲池，主要实现以下功能

- 从连接池获取或创建可用连接
- 使用完毕之后，把连接返还给连接池
- 在系统关闭前，断开所有连接并释放连接占用的系统资源
- 能够处理无效连接，并能够限制连接池中的连接总数不低于某个预定值和不超过某个预定值

最小连接数与最大连接数

- 最小连接数：连接池一直保持的数据库连接
  - 如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费
- 最大连接数：连接池能申请的最大连接数
  - 如果数据库连接请求超过次数，后面的数据库连接请求将被加入到等待队列中，会影响后续的数据库操作

如果最小连接数与最大连接数相差很大，那么最先连接请求将会获利，之后超过最小连接数量的连接请求等价于建立一个新的数据库连接。不过，这些大于最小连接数的数据库连接在使用完不会马上被释放，他将被放到连接池中等待重复使用或是空间超时后被释放

数据库连接池模拟

```
// 需要实现DataSource接口
// 需要注意数据库连接池要保证线程安全
// LinkedList的添加和删除操作效率高
private static LinkedList<Connection> pool = (LinkedList<Connection>)
Collections.synchronizedList(new LinkedList<Connection>());

static {
    try {
        // 向连接池中放10个连接
        for (int i = 0; i < 10; i++) {
            Connection con = DBUtil.getConnection();
            pool.add(con);
        }
    } catch (SQLException e) {
        throw new ExceptionInInitializerError("初始化数据库连接失败！");
    }
}

public static Connection getConnectionFromPool() {
    Connection con = null;
    if (pool.size() > 0) {
        // 将连接池中的一个连接取出
        con = pool.removeFirst();
        return con;
    } else {
        throw new RuntimeException("无空闲连接");
    }
}

// 当程序用完连接后，需要将该连接重新放入到连接池中
public static void release(Connection con) {
```

```
    pool.addLast(con);
}
```

第三方连接池

在实际应用中，通常不需要我们自己编写数据库连接池，有很多组织都提供了数据库连接池

- c3p0：开源的，成熟的，高并发第三方数据库连接池，相关的文档资料比较完善
- DBCP：DataBase Connection Pool，由 Apache 开发的一个数据库连接池，性能不太好，Apache 又开发了 Tomcat JDBC Pool来替代 DBCP
- Druid：阿里巴巴出品，号称是 Java 语言中最好的数据库连接池，能够提供强大的监控和扩展功能
- HiKariCP：日语中光的意思，性能很好，非常轻巧

Druid

可以使用 properties 或者 XML 配置

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/test
username=root
password=1234
#初始化的连接个数
initialSize=10
#最大连接数
maxActive=20
#最小连接数
minIdle=10
```

DruidUtil

```
// 得到一个Druid的数据源
private static DruidDataSource dataSource = null;

static {
    Properties properties = new Properties();

    try {
        //加载配置文件
 properties.load(DruidUtil.class.getClassLoader().getResourceAsStream("db.properties")
);
        //得到一个数据源
        dataSource =
(DruidDataSource)DruidDataSourceFactory.createDataSource(properties);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

```
// 从数据源中得到一个连接对象
// 这个返回的Connection实际上是Druid经过装饰之后的Cconnection
public static Connection getConnection() {
    try {
        return dataSource.getConnection();
    } catch (SQLException e) {
        throw new RuntimeException("服务器错误");
    }
}
```