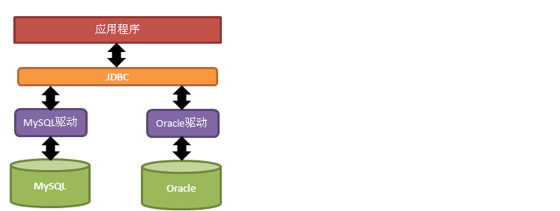
JDBC基础

JDBC(Java Database Connectivity)代表Java编程语言与数据库连接的标准API,然而JDBC只是接口,JDBC驱动才是真正的接口实现,没有驱动无法完成数据库连接. 每个数据库厂商都有自己的驱动,用来连接自己公司的数据库(如Oricle, MySQL, DB2, MS SQLServer).



下面我们以MySQL为例,JDBC编程大致步骤如下:

|  |
| --- |
| <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.36</version>  </dependency>  #db.properties  jdbc.driver=com.mysql.jdbc.Driver jdbc.url=jdbc:mysql://127.0.0.1:3306/mybatis jdbc.user=root jdbc.password=123456  public static Connection getConnection(String file) {  try {  Properties properties = new Properties();  properties.load(ClassLoader.*getSystemResourceAsStream*(file));  Class.*forName*(properties.getProperty("jdbc.driver"));  String url = properties.getProperty("jdbc.url");  String username = properties.getProperty("jdbc.user");  String password = properties.getProperty("jdbc.password");  return DriverManager.*getConnection*(url, username, password);  } catch (Exception e) {  e.printStackTrace();  return null;  } }  @Test public void test() throws Exception {  //1. 获取连接  Connection connection = *getConnection*("db.properties");  //2. 通过 Statement 执行SQL  Statement statement = connection.createStatement();  //3. 执行sql，返回结果集  ResultSet result = statement.executeQuery("SELECT \* FROM employee");  int columnCount = result.getMetaData().getColumnCount();  //4. 处理结果集  while (result.next()) {  for (int i = 1; i <= columnCount; ++i) {  System.*out*.printf("%s\t", result.getObject(i));  }  System.*out*.println();  }  statement.close();  connection.close(); } |

# 数据库连接

## 获取连接(DriverManger)

|  |
| --- |
| **#JDBC规定: 驱动类在被加载时,需要主动把自己注册到DriverManger中:)**  public class Driver extends NonRegisteringDriver implements java.sql.Driver {  public Driver() throws SQLException {  }  static {  try {  DriverManager.registerDriver(new Driver());  } catch (SQLException var1) {  throw new RuntimeException("Can't register driver!");  }  }  代码显示:只要去加载com.mysql.jdbc.Driver类那么就会执行static块, 从而把com.mysql.jdbc.Driver注册到DriverManager中.  j**ava.sql.DriverManager是用于管理JDBC驱动的服务类,其主要功能是获取Connection对象:**   * static Connection getConnection(String url, Properties info) * static Connection getConnection(String url, String user, String password)   另: 还可以在获取Connection的URL中设置参数,如: jdbc:mysql://host:port/database？useUnicode=true&characterEncoding=UTF8  useUnicode=true&characterEncoding=UTF8指定连接数据库的过程中使用Unicode字符集/UTF-8编码; |

## Connection API

java.sql.Connection代表数据库连接,每个Connection代表一个物理连接会话, 该接口提供如下创建Statement的方法, 只有获取Statement之后才可执行SQL语句：

|  |  |
| --- | --- |
| 方法 | 描述 |
| Statement createStatement() reparedStatement prepareStatement(String sql) CallableStatement prepareCall(String sql) | 创建不同statement对象 |
| void setTransactionIsolation(int level)  void setAutoCommit(boolean autoCommit)  void rollback()  void rollback(Savepoint savepoint) void commit() | 设置事务的隔离级别，是否自动提交事务，事务回滚和事务提交 |
| DatabaseMetaData getMetaData() | 返回数据库元信息 |

## DatabaseMetaData类

DatabaseMetaData 类中提供了许多方法用于获得数据源的各种信息，通过这些方法可以非常详细的了解数据库的信息：

|  |  |
| --- | --- |
| 方法 | 描述 |
| getURL() | 返回数据库的URL |
| getUserName() | 返回连接数据库当前用户名 |
| isReadOnly() | 是否只允许读操作。 |
| getDatabaseProductName() | 返回数据库的产品名称 |
| getDatabaseProductVersion() | 返回数据库的版本号 |
| getDriverName() | 返回驱动驱动程序的名称 |
| getDriverVersion() | 返回驱动程序的版本号 |

## 数据库连接池

前面通过DriverManger获得Connection, 一个Connection对应一个实际的物理连接,每次操作都需要打开物理连接, 使用完后立即关闭;这样频繁的打开/关闭连接会造成不必要的数据库系统性能消耗.。

数据库连接池提供的解决方案是:当应用启动时,主动建立足够的数据库连接,并将这些连接组织成连接池,每次请求连接时,无须重新打开连接,而是从池中取出已有连接,使用完后并不实际关闭连接,而是归还给池.

JDBC数据库连接池使用javax.sql.DataSource表示, DataSource只是一个接口, 其实现通常由服务器提供商(如WebLogic, WebShere)或开源组织(如DBCP,C3P0和HikariCP)提供.数据库连接池的常用参数如下:

* 数据库初始连接数;
* 连接池最大连接数;
* 连接池最小连接数;
* 连接池每次增加的容量;

### C3P0

Tomcat默认使用的是DBCP连接池,但相比之下,C3P0则比DBCP更胜一筹(Hibernate推荐使用C3P0),C3P0不仅可以自动清理不再使用的Connection, 还可以自动清理Statement/ResultSet, 使用C3P0需要在pom.xml中添加如下依赖:

|  |
| --- |
| <dependency>   <groupId>com.mchange</groupId>   <artifactId>c3p0</artifactId>   <version>0.9.5.2</version>  </dependency>  <dependency>   <groupId>com.mchange</groupId>   <artifactId>mchange-commons-java</artifactId>   <version>0.2.11</version>  </dependency> |
| private static DataSource *dataSource*;  public static DataSource getDataSource(String file) {  if (*dataSource* == null) {  synchronized (JdbcTest.class) {  if (*dataSource* == null) {  try {  Properties properties = new Properties();  properties.load(ClassLoader.*getSystemResourceAsStream*(file));  ComboPooledDataSource source = new ComboPooledDataSource();  source.setDriverClass(properties.getProperty("jdbc.driver"));  source.setJdbcUrl(properties.getProperty("jdbc.url"));  source.setUser(properties.getProperty("jdbc.user"));  source.setPassword(properties.getProperty("jdbc.password"));  // 设置连接池最大连接数  source.setMaxPoolSize(Integer.*valueOf*(properties.getProperty("pool.max.size")));  // 设置连接池最小连接数  source.setMinPoolSize(Integer.*valueOf*(properties.getProperty("pool.min.size")));  // 设置连接池初始连接数  source.setInitialPoolSize(Integer.*valueOf*(properties.getProperty("pool.init.size")));  // 设置最大空闲时间  source.setMaxIdleTime(Integer.*valueOf*(properties.getProperty("pool.max.idle\_time")));  *dataSource* = source;  } catch (Exception e) {  throw new RuntimeException(e);  }  }  }  }  return *dataSource*; } |

# SQL执行

## Statement接口

java.sql.Statement可用于执行DDL/DML/DCL语句:

|  |  |
| --- | --- |
| 方法 | 描述 |
| boolean execute(String sql) | 可以用于执行所有的SQL语句 |
| ResultSet executeQuery(String sql) | 执行查询返回resultSet |
| int executeUpdate(String sql) | 执行DDL DML语句 |
| ResultSet getResultSet() | 获取execute方法返回结果集 |
| int getUpdateCount() | 获取execute方法(DML)影响行数 |

Java 1.7还新增了closeOnCompletion()方法,当所有依赖于当前Statement的ResultSet关闭时,该Statement自动关闭.

|  |
| --- |
| Statement使用executeUpdate方法执行DDL/DML(不包含select)语句:执行DDL该方法返回0; 执行DML返回受影响的记录数.。  @Test public void test3() throws Exception {  Connection connection = *getConnection*("db.properties");  Statement statement = connection.createStatement();  int res = statement.executeUpdate("CREATE TABLE t\_ddl(id **INT** auto\_increment PRIMARY KEY) " );  System.*out*.println(res);  }  @Test public void test() throws Exception {  Connection connection = *getConnection*("db.properties");  Statement statement = connection.createStatement();  ResultSet employees = statement.executeQuery("SELECT \* FROM employee");  int columnCount = employees.getMetaData().getColumnCount();  while (employees.next()) {  for (int i = 1; i <= columnCount; ++i) {  System.*out*.printf("%s\t", employees.getObject(i));  }  System.*out*.println();  }  statement.close();  connection.close(); } |

## **PreparedStatement接口**

PreparedStatement是Statement的子接口, 它可以预编译SQL语句,编译后的SQL模板被存储在PreparedStatement对象中,每次使用时首先为SQL模板设值,然后执行该语句(因此使用PreparedStatement效率更高).创建PreparedStatement需要使用Connection的prepareStatement(String sql)方法,该方法需要传入SQL模板,可以包含占位符参数:

PreparedStatement statement = connection.prepareStatement("INSERT INTO t\_ddl(username, password) VALUES (?, ?)")

PreparedStatement也提供了excute等方法来执行SQL语句, 只是这些方法无须传入参数, 因为SQL语句已经存储在PreparedStatement对象中.

由于执行SQL前需要为SQL模板传入参数值,PreparedStatement提供了一系列的setXxx(int parameterIndex, X x)方法;另外,如果不清楚SQL模板各参数的类型,可以使用setObject(int parameterIndex, Object x)方法传入参数, 由PreparedStatement来负责类型转换.

|  |
| --- |
| @Test public void test4() throws Exception {  Connection connection = null;  PreparedStatement ps = null;  try {  connection = *getConnection*("db.properties");  ps = connection.prepareStatement("SELECT \* FROM employee where id=?");  ps.setLong(1,1l);  ResultSet result = ps.executeQuery();  while (result.next()) {  System.*out*.print(result.getLong(1) + "\t");  System.*out*.print(result.getString(2) + "\t");  System.*out*.print(result.getString(3) + "\t");  System.*out*.print(result.getString(4));  System.*out*.println();  }  }catch (Exception e){  e.printStackTrace();   }finally {  ps.close();  connection.close();  } }  注意:  1. SQL语句的占位符参数只能代替普通值, 不能代替表名/列名等数据库对象, 更不能代替INSERT/SELECT等关键字.使用PreparedStatement还有另外一2. 使用PreparedStatement无须拼接SQL字符串,因此可以防止SQL注入:  3. 默认使用PreparedStatement是没有开启预编译功能的,需要在URL中给出useServerPrepStmts=true参数来开启此功能;  4. 当使用不同的PreparedStatement对象来执行相同SQL语句时,还是会出现编译两次的现象,这是因为驱动没有缓存编译后的函数key,导致二次编译.如果希望缓存编译后的函数key,那么就要设置cachePrepStmts=true参数.  5. 设置预编译缓存的大小:cachePrepStmts=true&prepStmtCacheSize= 50&prepStmtCacheSqlLimit=300`  jdbc:mysql://host:port/database?useServerPrepStmts=true&cachePrepStmts=true&prepStmtCacheSize=50&prepStmtCacheSqlLimit=300 |

## CallableStatement接口

调用存储过程需要使用CallableStatement,可以通过Connection的prepareCall()方法来创建,创建时需要传入调用存储过程的SQL语句,形式为:

{CALL procedure\_name(?, ?, ?)}

存储过程的参数既有入参,也有回参;

入参可通过setXxx(int parameterIndex/String parameterName, X x);

回参通过调用registerOutParameter(int parameterIndex, int sqlType), 经过上面步骤, 就可以调用execute()方法来调用该存储过程, 执行结束, 则通过getXxx(int parameterIndex/String parameterName)方法获取指定回参的值:

|  |
| --- |
| @Test public void test6() throws Exception {  Connection connection = *getConnection*("db.properties");  CallableStatement cs = connection.prepareCall("{CALL countEmployee(?,?)}");  cs.setString(1,"test02");  cs.registerOutParameter(2,Types.*INTEGER*);  cs.execute();  Integer output = cs.getInt(2);  System.*out*.println(output); } |

# 操作结果集

JDBC使用ResultSet封装查询结果,然后通过ResultSet的记录指针来读取/更新记录.并提供了ResultSetMetaDate来获得ResultSet对象的元数据信息.

## ResultSet

java.sql.ResultSet是结果集对象,可以通过列索引/列名来读/写数据, 它提供了如下常用方法来移动记录指针:

|  |  |
| --- | --- |
| **方法** | **描述** |
| boolean next() | 向后移动指针，如果有值返回true |
| getXXX(String columnLabel)  getXXX(int int columnIndex) | 通过列名或者列索引（从1开始）获取查询列值XXX对应其类型 |
| ResultSetMetaData getMetaData() | 获取结果集元信息 |

## ResultSetMetaData

ResultSet提供了getMetaData()方法来获取ResultSetMetaData以分析关于ResultSet的描述信息(前面我们已经使用ResultSetMetaData来获取结果集的列数以及列名):

|  |  |
| --- | --- |
| 方法 | 描述 |
| getColumnName(int column) | 获取指定列的名称 |
| getColumnCount() | 返回当前 ResultSet 对象中的列数。 |
| getColumnTypeName(int column) | 检索指定列的数据库特定类型名称。 |
| getColumnDisplaySize(int column) | 指定列的最大标准宽度(字符) |
| isNullable(int column) | 指定列中的值是否可以为 null。 |
| isAutoIncrement(int column) | 是否自动为指定列进行编号 |

# 事务

事务是由一步/几步数据库操作序列组成的逻辑执行单元, 这些操作要么全部执行, 要么全部不执行. MySQL事务功能需要有InnoDB存储引擎的支持, 详见MySQL存储引擎InnoDB与Myisam的主要区别.事务的

ACID特性：

* 原子性(A: Atomicity): 事务是不可再分的最小逻辑执行体;
* 一致性(C: Consistency): 事务执行的结果, 必须使数据库从一个一致性状态, 变为另一个一致性状态.
* 隔离性(I: Isolation): 各个事务的执行互不干扰, 任意一个事务的内部操作对其他并发事务都是隔离的(并发执行的事务之间不能看到对方的中间状态,不能互相影响)
* 持续性(D: Durability): 持续性也称持久性(Persistence), 指事务一旦提交, 对数据所做的任何改变都要记录到永久存储器(通常指物理数据库).

Commit/Rollback，当事务所包含的全部操作都成功执行后提交事务,使操作永久生效,事务提交有两种方式:

* 显式提交: 使用commit;
* 自动提交: 执行DDL/DCL语句或程序正常退出;

当事务所包含的任意一个操作执行失败后应该回滚事务, 使该事务中所做的修改全部失效, 事务回滚也有两种方式:

* 显式回滚: 使用rollback;
* 自动回滚: 系统错误或强行退出.

注意: 同一事务中所有的操作,都必须使用同一个Connection.

|  |
| --- |
| @Test public void test2() throws Exception {  Connection connection = *getConnection*("db.properties");  boolean autoCommitFlag = connection.getAutoCommit();  System.*out*.println("自动提交：" + autoCommitFlag);  // 关闭自动提交, 开启事务  connection.setAutoCommit(false);  PreparedStatement ps = null;  try{  ps = connection.prepareStatement("UPDATE `employee` SET `last\_name`=? WHERE `id`=?");  ps.setString(1,"haha");  ps.setLong(2,1l);  ps.executeUpdate();  if (true) throw new RuntimeException("报错啦");  connection.commit();  }catch (Exception e){  connection.rollback();  ps.close();  connection.close();  e.printStackTrace();  } } |

# 批处理

多条SQL语句被当做同一批操作同时执行.调用Statement对象的addBatch(String sql)方法将多条SQL语句收集起来, 然后调executeBatch()同时执行.为了让批量操作可以正确进行, 必须把批处理视为单个事务, 如果在执行过程中失败, 则让事务回滚到批处理开始前的状态.

|  |
| --- |
| @Test public void test7() throws Exception {  Connection connection = *getConnection*("db.properties");  connection.setAutoCommit(false);  Statement statement = connection.createStatement();  for (int i = 0; i < 10; ++i) {  //添加批处理语句  statement.addBatch("insert into employee(last\_name,gender,email) values(MD5(RAND() \* 10000) , 'm','h861336327@163.com')");  }  //执行批处理语句  statement.executeBatch();  connection.commit();  statement.close();  connection.close(); }  注:对于批处理,也可以使用PreparedStatement,建议使用Statement,因为PreparedStatement的预编译空间有限,当数据量过大时,可能会引起内存溢出.  MySQL默认也没有打开批处理功能,需要在URL中设置rewriteBatchedStatements=true参数打开. |