作为一门强大的变成语言，groovy提供了比java更加方便直观的通过JDBC访问数据库的语言支持。groovy对数据库的访问能力都包含在groovy.sql库中，这个库只有四个类：Sql、DataSet、GroovyResultSet和GroovyRowResult。下图说明了groovy sql的api栈：



图1

groovy能够提供比其他脚本语言更友好的JDBC编码方式，同样groovy在对象层面上也能很好的访问JDBC。本章中有一些代码使用了DTO和DAO之类的数据库相关的设计模式对象。同样你也会发现使用groovy进行数据库编程，很少需要去创建这些DTO或者DAO对象，这大大减少了日常编程工作。

# 基本数据库操作

基本数据库操作包括：create、read、update和delete，简称CRUD。

## 配置访问数据库

不管使用何种技术，要访问一个数据库必须需要提供下面四个信息：

* 数据库的URL
* 用户名
* 密码
* 数据库驱动class name

数据库URL是一种平台无关的定位数据的方式，它往往由jdbc:开头，后面跟和数据库相关的协议名。加入访问oracle数据库：

|  |
| --- |
| jdbc:oracle:thin:@10.100.133.105:1522:d0tcs |

驱动名为：

|  |
| --- |
| oracle.jdbc.driver.OracleDriver |

注意需要将oracle的驱动包ojdbc6.jar放到groovy的classpath下。

有了上述四个信息就可以访问到一个数据库：

|  |
| --- |
| import groovy.sql.Sql  db = Sql.newInstance('jdbc:oracle:thin:@10.100.133.105:1522:d0tcs',  'tf',  '',  'oracle.jdbc.driver.OracleDriver') |

### DriverManager对比DataSource

如果回头再看看图1，你会发现在JDBC API下有两个概念：JDBC DriverManager和DataSource。使用Sql.newInstance方法使用的就是DriverManager这种方式，DriverManager方式可以看成是一种古老的底层的连接数据库的方式。

尽管JDBC仍然支持DriverManager，但是使用DataSource是一种更被推荐的方式。除了提供数据库连接以外，DataSource还可以维持一个数据库连接池并能支持分布式事务。数据库连接池能提供更好的性能，并且groovy SQL能自动将使用完毕的连接返回到池中。在应用服务器上，往往由服务器提供dataSource，应用通过JNDI获取dataSource。

获得dataSorce以后，就可以作为Sql对象的构造函数参数传入：

|  |
| --- |
| import groovy.sql.Sql  dataSource = ....//get dataSource  db = new Sql(dataSource) |

如果当前已经获取了数据库连接，也可以使用new Sql(connection)方法获取Sql实例。在这种情况下，因为groovy Sql并没有负责创建connection对象，所以它并不负责关闭数据库连接，需要自己在代码中调用connection.close()方法。

如果当前已经有了一个Sql实例，则可以用new Sql(db)这个方法克隆一个Sql实例。

总之，获取了Sql实例之后，它就代表着一个数据库连接，就可以用到来执行sql语句了。

## 执行sql

如果你当前有个变量名为db的Sql对象实例，则可以使用如下方法执行sql：

|  |
| --- |
| db.execute(statement) |

这个方法执行的时候，groovy SQL会做如下工作：获取数据库连接，构造和配置sql语句，发送语句给数据库执行，如有异常时记录异常日志，执行关闭关闭资源（statement和connection）。

### 执行DDL

利用execute方法可以很方便的执行DDL操作，例如：

|  |
| --- |
| db.execute '''  CREATE TABLE Athlete (  athleteId INTEGER GENERATED BY DEFAULT AS IDENTITY,  firstname VARCHAR(64),  lastname VARCHAR(64),  dateOfBirth DATE  );  CREATE INDEX athleteIdx ON Athlete (athleteId);  ''' |

### 执行DML

插入数据：

|  |
| --- |
| db.execute '''  INSERT INTO Athlete (firstname, lastname, dateOfBirth)  VALUES ('Paul', 'Tergat', '1969-06-17');  INSERT INTO Athlete (firstname, lastname, dateOfBirth)  VALUES ('Khalid', 'Khannouchi', '1971-12-22');  INSERT INTO Athlete (firstname, lastname, dateOfBirth)  VALUES ('Ronaldo', 'da Costa', '1970-06-07');  ''' |

执行prepared statement:

|  |
| --- |
| String athleteInsert = '''  INSERT INTO Athlete (firstname, lastname, dateOfBirth)  VALUES (?, ?, ?);  '''  db.execute athleteInsert, [  'Paul',  'Tergat',  '1969-06-17'  ]  db.execute athleteInsert, [  'Khalid',  'Khannouchi',  '1971-12-22'  ]  db.execute athleteInsert, [  'Ronaldo',  'da Costa',  '1970-06-07'  ] |

也可以使用GString来执行sql：

|  |
| --- |
| def athletes = [  [first: 'Paul', last: 'Tergat', birth: '1969-06-17'],  [first: 'Khalid', last: 'Khannouchi', birth: '1971-12-22'],  [first: 'Ronaldo', last: 'da Costa', birth: '1970-06-07']  ]  athletes.*each* { athlete ->  db.execute """  INSERT INTO Athlete (firstname, lastname, dateOfBirth)  VALUES (${athlete.first}, ${athlete.last}, ${athlete.birth});  """  } |

如果想要查看具体的sql语句执行情况，则可以打开groovy.sql包的日志：

|  |
| --- |
| import groovy.sql.Sql  import java.util.logging.Level  import java.util.logging.Logger  Logger.*getLogger*('groovy.sql').level = Level.*FINE* |

更新和删除数据的方式和插入数据相同：

|  |
| --- |
| db.execute '''  DELETE FROM Athlete WHERE firstname = 'Dierk';  '''  db.execute '''  INSERT INTO Athlete (firstname, lastname, dateOfBirth)  VALUES (?, ?, ?);  ''', ['Dirk', 'Koenig', '1968-04-19']  String wrong = 'Dirk'  String right = 'Dierk'  db.execute """  UPDATE Athlete SET firstname = $right WHERE firstname = $wrong;  """ |

execute方法返回一个boolean值表示语句是否成功，对于更新操作还有一个executeUpdate方法，该方法返回一个数值，用于表示有多少条记录被更新。下表说明了execute方法的不同版本：

|  |  |  |
| --- | --- | --- |
| 返回类型 | 方法名 | 参数 |
| boolean | execute | String statement |
| boolean | execute | String prepStmt, List values |
| boolean | execute | Gstring prepStmt |
| int | executeUpdate | String statement |
| Int | executeUpdate | String prepStmt, List values |
| int | executeUpdate | GString prepStmt |

### 查询数据

查询数据需要返回具体的查询结果，显然execute方法并不适用与查询，Sql类提供下面的方法用于查询：

|  |  |  |
| --- | --- | --- |
| 返回类型 | 方法 | 参数 |
| void | eachRow | String statement {row -> code} |
| void | eachRow | String statement, List values {row -> code} |
| void | eachRow | GString prepStmt {row - > code} |
| void | query | String statement,{resultSet -> code} |
| void | query | String prepStmt, List values {resultSet -> code} |
| Void | query | GString prepStmt {resultSet -> code} |
| List | rows | String statement |
| List | rows | String prepStmt, List values |
| Object | firstRow | String statement |
| Object | firstRow | String prepStmt, List values |

方法eachRow和query使用一个closure来处理查询结果，query调用closure一次并传入整个结果集，eacheRow遍历返回的结果集，每次处理一条记录。

#### 使用eachRow遍历查询结果

使用eachRow closure的参数对应当前记录，可以将closure的参数看成是一个对象，对象的属性对应记录的字段：

|  |
| --- |
| db.eachRow('select id, name from WORKFLOW\_STEP') { step ->  println step.name  } |

也可以将closure的参数看成是一个数组:

|  |
| --- |
| db.eachRow('select id, name from WORKFLOW\_STEP') { step ->  println step[1]  } |

事实上，传入closure的参数是一个groovy.sql.GroovyResultSet对象。

#### 使用query获取查询的结果集

query方法的closure的参数是一个结果集，这样就可以控制结果集的遍历过程，例如只获取第二条记录：

|  |
| --- |
| db.query('select id, name from WORKFLOW\_STEP') { resultSet ->  int i = 0  while (resultSet.next()) {  i ++  if (i == 2 ) {  println resultSet.getString('name')//or resultSet.getString(2)  break  }  }  } |

query方法没有eachRow方法灵活，你需要调用resultSet的next方法来遍历结果集，而且还需要调用字段类型相关的get方法（例如getString，getDate等），而且字段的index不是从0开始而是从1开始的。

#### 一次性获取所有的查询结果记录

使用rows方法可以获取查询结果记录，该方法返回一个List，列表中的每个元素都是代表一个查询记录的对象，可以像在eachRow方法中那样的使用。

上面使用query方法的例子可以写成如下：

|  |
| --- |
| def steps = db.rows('select id, name from WORKFLOW\_STEP')  println steps[1].name // or steps[1][1] |

# 在grails中使用groovy SQL

grails框架提供了强大的GORM，但是有些情况下可能只能够直接使用sql来访问数据库。这种情况下，我们也可以在Grails框架中直接使用groovy SQL。

每一个domain class都有一个静态的withSession方法，在该方法的closure里，我们可以通过hibernate session获取当前的数据库连接，然后再创建groovy的Sql对象:

|  |
| --- |
| Author.*withSession* { s ->  s.doWork new Work(){  void execute(Connection c) {  def db = new Sql(c)  db.eachRow('') {  //....  }  }  }  } |