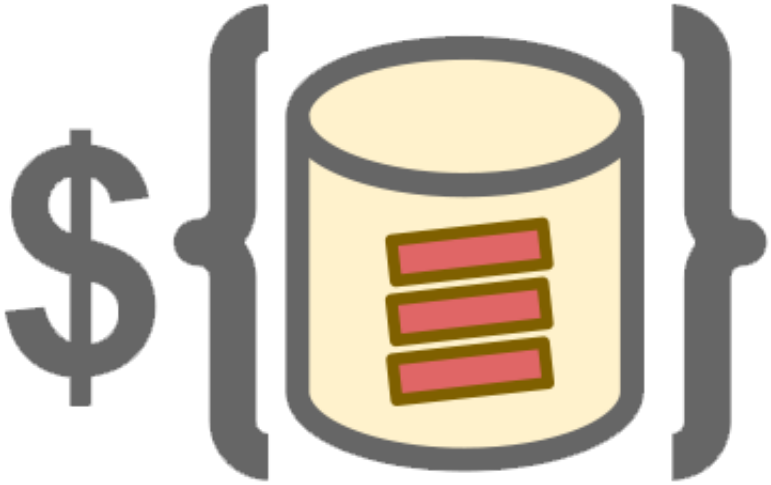
ScalikeJDBC操作API

《ScalikeJDBC：基于SQL的简洁DB访问类库》文章中已经介绍了ScalikeJDBC到底是个什么东西。本文将介绍ScalikeJDBC的常用操作（Operations）API。



查询API

ScalikeJDBC中有多种查询API，包括single, first, list 和foreach，他们内部都是调用java.sql.PreparedStatement#executeQuery()实现的。下面将分别介绍如何使用这个API。

single查询

single函数返回匹配到的单行数据，并且封装成Option值。如果single函数匹配到多行，那么在运行的时候会抛出异常。使用single函数如下：

```
import scalikejdbc._

val id = 123

// simple example
val name: Option[String] = DB readOnly { implicit session =>
  sql"select name from emp where id = ${id}".map(rs => rs.string("name")).single.apply()
}

// defined mapper as a function
val nameOnly = (rs: WrappedResultSet) => rs.string("name")
val name: Option[String] = DB readOnly { implicit session =>
  sql"select name from emp where id = ${id}".map(nameOnly).single.apply()
}
```

```
// define a class to map the result
case class Emp(id: String, name: String)
val emp: Option[Emp] = DB readOnly { implicit session =>
  sql"select id, name from emp where id = ${id}"
    .map(rs => Emp(rs.string("id"), rs.string("name"))).single.apply()
}

// QueryDSL
object Emp extends SQLSyntaxSupport[Emp] {
  def apply(e: ResultName[Emp])(rs: WrappedResultSet): Emp = new Emp(id = rs.get(e.id), name
 = rs.get(e.name))
}
val e = Emp.syntax("e")
val emp: Option[Emp] = DB readOnly { implicit session =>
  withSQL { select.from(Emp as e).where.eq(e.id, id) }.map(Emp(e.resultName)).single.apply()
}
```

关于上面的QueryDSL详细使用，可以参见http://scalikejdbc.org/documentation/query-dsl.html。

返回多行结果中的第一行

first函数返回多行结果中的第一行结果，而且返回的类型也是Option封装的。

```
val name: Option[String] = DB readOnly { implicit session =>
  sql"select name from emp".map(rs => rs.string("name")).first.apply()
}

val e = Emp.syntax("e")
val name: Option[String] = DB readOnly { implicit session =>
  withSQL { select(e.result.name).from(Emp as e) }.map(_.string(e.name)).first.apply()
}
```

返回List的结果

list函数将匹配到的多行存储在scala.collection.immutable.List中：

```
val name: List[String] = DB readOnly { implicit session =>
  sql"select name from emp".map(rs => rs.string("name")).list.apply()
```

```
}

val e = Emp.syntax("e")
val name: Option[String] = DB readOnly { implicit session =>
  withSQL { select(e.result.name).from(Emp as e) }.map(_.string(e.name)).list.apply()
}
```

Foreach操作

foreach函数允许你在iterations中进行一些有副作用的计算，这个函数在ResultSet含有大量的返回值情况下特别有用。

```
DB readOnly { implicit session =>
  sql"select name from emp".foreach { rs =>
    out.write(rs.string("name"))
  }
}

val e = Emp.syntax("e")
DB readOnly { implicit session =>
  withSQL { select(e.name).from(Emp as e) }.foreach { rs =>
    out.write(rs.string(e.name))
  }
}
```

微信扫一扫，加关注
即可及时了解Spark、Hadoop或者Hbase等相关的文章
欢迎关注微信公共帐号：iteblog_hadoop

过往记忆博客(http://www.iteblog.com)专注于Hadoop、Spark、Flume、Hbase等技术的博客，欢迎关注。

Hadoop、Hive、Hbase、Flume等交流群：138615359和149892483

如果想及时了解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

设置JDBC fetchSize

PostgreSQL的JDBC驱动默认情况下（fetchSize=0）将无限制地获取返回的结果，这种情况会导致内存相关的问题：http://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html#setFetchSize-int-。但是在ScalikeJDBC 2.0.5之后，我们可以设置JDBC的fetchSize值：

```
val e = Emp.syntax("e")
DB readOnly { implicit session =>
  sql"select name from emp"
    .fetchSize(1000)
    .foreach { rs => out.write(rs.string("name")) }
}
```

或者直接在scalikejdbc.DBSession上设置fetchSize：

```
val (e, c) = (Emp.syntax("e"), Cmp.syntax("c"))

DB readOnly { implicit session =>
  session.fetchSize(1000)

  withSQL { select(e.name).from(Emp as e) }.foreach { rs =>
    out.write(rs.string(e.name)
  }
  withSQL { select(c.name).from(Cmp as c) }.foreach { rs =>
    out.write(rs.string(c.name))
  }
}
```

实现自定义的抽取器(Extractor)

在某些情况下，你可能需要实现自定义的抽取器，这在测试我们查询后的结果非常有用，下面的例子将展示如何在结果集中保留null值。

```
def toMap(rs: WrappedResultSet): Map[String, Any] = {
  (1 to rs.metaData.getColumnCount).foldLeft(Map[String, Any]()) { (result, i) =>
    val label = rs.metaData.getColumnLabel(i)
    Some(rs.any(label)).map { nullableValue => result + (label -> nullableValue) }.getOrElse(resul
```

```
t)
  }
}

sql"select * from emp".map(rs => toMap(rs)).single.apply()
```

使用ParameterBinder

ParameterBinder[A]使得我们可以在ScalikeJDBC中自定义如何将参数和PreparedStatement进行绑定。下面的例子将展示如何在InputStream和PreparedStatement进行绑定的情况使用ResultSet#setBinaryStream：

```
sql"create table blob_example (id bigint, data blob)").execute.apply()

val bytes = scala.Array[Byte](1, 2, 3, 4, 5, 6, 7)

val bytesBinder = ParameterBinder[InputStream](
  value = new ByteArrayInputStream(bytes),
  binder = (stmt: PreparedStatement, idx: Int) => stmt.setBinaryStream(idx, in, bytes.length)
)

sql"insert into blob_example (data) values (${bytesBinder})").update.apply()
```

使用TypeBinder

TypeBinder[A]是一种类型类，它主要的功能就是从Result中抽取类型为A的值，下面的例子将展示如何将long类型的值MemberId变量进行绑定：

```
import scalikejdbc._
import java.sql.ResultSet

implicit val session = AutoSession

// prepare data
sql"create table member (id bigint not null primary key, name varchar(100))".execute.apply()
sql"insert into member values (1, 'Alice')".update.apply()
sql"insert into member values (2, 'Bob')  ".update.apply()
sql"insert into member values (3, 'Chris')".update.apply()

case class MemberId(id: Long)
```

```
case class Member(id: MemberId, name: String)

// compilation error here
val ids: Seq[MemberId] = sql"select * from member".map(_.get[MemberId]("id")).list.apply()

// <console>:24: error: could not find implicit value for evidence parameter of type scalikejdbc.
TypeBinder[MemberId]
//        val ids: Seq[MemberId] = sql"select * from member".map(_.get[MemberId]("id")).list.ap
ply()
//
```

默认情况下MemberId类肯定是不能支持的。所以我们需要为MemberId类自定义TypeBinder：

```
// define TypeBinder[MemberId] for implicit value
implicit val memberIdTypeBinder: TypeBinder[MemberId] = new TypeBinder[MemberId] {
  def apply(rs: ResultSet, label: String): MemberId = MemberId(rs.getLong(label))
  def apply(rs: ResultSet, index: Int): MemberId = MemberId(rs.getLong(index))
}

val ids: Seq[MemberId] = sql"select id from member".map(_.get[MemberId]("id")).list.apply()
// ids: Seq[MemberId] = List(MemberId(1), MemberId(1), MemberId(2), MemberId(3))
```

当然，我们在这种场景下完全可以不定义TypeBinder，可以用下面方式实现：

```
val ids: Seq[MemberId] = sql"select id from member".map(rs => MemberId(rs.get("id"))).list.ap
ply()
```

使用Java SE 8 Date Time API (JSR-310)

ScalikeJDBC仍然支持Java SE 7。所以在ScalikeJDBC中提供了Java 8 Date Time API可选支持：

```
libraryDependencies += "org.scalikejdbc" %% "scalikejdbc-jsr310" % "2.3.5"
```

使用它也是很简单的，仅仅需要引入import scalikejdbc.jsr310._:

```
import scalikejdbc._, jsr310._
import java.time._

case class Group(
  id: Long,
  name: Option[String],
  createdAt: ZonedDateTime)

object Group extends SQLSyntaxSupport[Group] {
  def apply(g: SyntaxProvider[Group])(rs: WrappedResultSet): Group = apply(g.resultName)(rs)
  def apply(g: ResultName[Group])(rs: WrappedResultSet): Group = new Group(
    id = rs.get(g.id),
    name = rs.get(g.name),
    createdAt = rs.get(g.createdAt)
    // (or) createdAt = rs.get[ZonedDateTime](g.createdAt)
    // (or) createdAt = rs.zonedDateTime(g.createdAt)
  )
}
```

更新API

update最终运行的是java.sql.PreparedStatement#executeUpdate()

```
import scalikejdbc._

DB localTx { implicit session =>
  sql"""insert into emp (id, name, created_at) values (${id}, ${name}, ${DateTime.now})"""
    .update.apply()
  val id = sql"insert into emp (name, created_at) values (${name}, current_timestamp)"
    .updateAndReturnGeneratedKey.apply()
  sql"update emp set name = ${newName} where id = ${id}".update.apply()
  sql"delete emp where id = ${id}".update.apply()
}

val column = Emp.column
DB localTx { implicit s =>
  withSQL {
    insert.into(Emp).namedValues(
      column.id -> id,
      column.name -> name,
      column.createdAt -> DateTime.now)
```

```
  }.update.apply()

  val id: Long = withSQL {
    insert.into(Empy).namedValues(column.name -> name, column.createdAt -> sqls.currentTim
estamp)
  }.updateAndReturnGeneratedKey.apply()

  withSQL { update(Emp).set(column.name -> newName).where.eq(column.id, id) }.update.appl
y()
  withSQL { delete.from(Emp).where.eq(column.id, id) }.update.apply()
}
```

Execute API

execute最终运行的是java.sql.PreparedStatement#execute().

```
DB autoCommit { implicit session =>
  sql"create table emp (id integer primary key, name varchar(30))".execute.apply()
}

// QueryDSL doesn't support DDL yet.
```

批处理(Batch)API

batch和batchByName最终运行的是java.sql.PreparedStatement#executeBatch()

```
import scalikejdbc._

DB localTx { implicit session =>
  val batchParams: Seq[Seq[Any]] = (2001 to 3000).map(i => Seq(i, "name" + i))
  sql"insert into emp (id, name) values (?, ?)".batch(batchParams: _*).apply()
}

DB localTx { implicit session =>
  sql"insert into emp (id, name) values ({id}, {name})"
    .batchByName(Seq(Seq('id -> 1, 'name -> "Alice"), Seq('id -> 2, 'name -> "Bob")):_*)
    .apply()
}

val column = Emp.column
```

```
DB localTx { implicit session =>
  val batchParams: Seq[Seq[Any]] = (2001 to 3000).map(i => Seq(i, "name" + i))
  withSQL {
    insert.into(Emp).namedValues(column.id -> sqls.?, column.name -> sqls.?)
  }.batch(batchParams: _*).apply()
}
```

本文翻译自：http://scalikejdbc.org/documentation/operations.html

本博客文章除特别声明，全部都是原创！
原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接: 【】（）