JDBC&&Shapeless

原始代码

```
object Database {

  def query(sql: String, conn: Connection): List[List[String]] = {
    var stmt: Statement = null
    try {
      stmt = conn.createStatement()
      stmt.executeQuery(sql).rows
    } catch {
      case e: SQLException =>
        List()
    } finally {
      if (stmt != null) stmt.close()
    }
  }

  implicit class ResultSetOp(rs: ResultSet) {

    private val columnLength: Int = rs.getMetaData.getColumnCount

    def rows: List[List[String]] = {
      def loop(rs: ResultSet, res: List[List[String]]): List[List[String]] =
        if (!rs.next()) res
        else {
          val row = (1 to columnLength).map(rs.getString).toList
          loop(rs, res :+ row)
        }
      loop(rs, Nil)
    }
  }
}
```

查询对象

```
def conn: Connection = {
  Class.forName("org.h2.Driver")
  DriverManager.getConnection("jdbc:h2:~/test")
}

case class Student(id: Long, name: String, score: Int)

private def list2Student(li: List[String]): Student = {
  val id = li(0).toLong
  val name = li(1).toString
  val score = li(2).toInt
  Student(id, name, score)
}
```

查询对象

```scala
case class Book(id: Long, name: String, price: Int)

private def list2Book(li: List[String]): Book = {
  val id = li(0).toLong
  val name = li(1)
  val street = li(2).toInt
  Book(id, name, street)
}
```

样例类：Student
属性：id: Long, name: String, score: Int
转换函数：list2Student: List[String] => Student

样例类：Book
属性：id: Long, name: String, price: Int
转换函数：list2Book: List[String] => Book

查询代码1

```
import IOUtils._
val sql1 = s"SELECT * FROM student"
val rows1 = using(conn)(query(sql1, _))
val students: List[Student] = rows1.map(list2Student)
```

```
val sql2 = s"SELECT * FROM book"
val rows2 = using(conn)(query(sql2, _))
val books: List[Book] = rows2.map(list2Book)
```

查询Student需要转换函数list2Student: List[String] => Student
查询Book需要转换函数list2Book: List[String] => Book
那查询Teacher需要转换函数list2Teacher: List[String] => Teacher
查询A需要转换函数list2A: List[String] => A

对于任意类型A都需要一个转换函数list2A: List[String] => A
能否省略转换函数呢？

使用公共接口

```
*/
trait Mapper1[A] {

  def to(li: List[String]): A
}
```

```
object Student extends Mapper1[Student] {
  override def to(li: List[String]): Student = {
    val id = li(0).toLong
    val name = li(1).toString
    val score = li(2).toInt
    Student(id, name, score)
  }
}
```

```
object Book extends Mapper1[Book] {
  override def to(li: List[String]): Book = {
    val id = li(0).toLong
    val name = li(1)
    val price = li(2).toInt
    Book(id, name, price)
  }
}
```

查询代码2

```
val stidtents1: List[Student] = using(conn)(query(sql1, _)).map(Student.to)
val books1: List[Book] = using(conn)(query(sql2, _)).map(Book.to)
```

无转换函数list2Student、list2Book
但是仍需要函数Student.to和Book.to来完成List[String] => A的转换
不同的是转换函数定义在类型A对应object中
Student 继承Mapper[Student]，实现to: List[String] => Student
Book继承Mapper[Book]，实现to: List[String] => Book
这样只是转换函数换了一个地方定义
还有其他的方法吗？

使用类型类实例

```scala
object Mapper1 {

  def apply[A](implicit ma: Mapper1[A]): Mapper1[A] = ma

  def instance[A](f: List[String] => A): Mapper1[A] = new Mapper1[A] {
    override def to(li: List[String]): A = f(li)
  }
```

```scala
  implicit val studentMapper: Mapper1[Student] = instance { li =>
    val id = li(0).toLong
    val name = li(1).toString
    val score = li(2).toInt
    Student(id, name, score)
  }

  implicit val addressMapper: Mapper1[Book] = instance { li =>
    val id = li(0).toLong
    val name = li(1)
    val price = li(2).toInt
    Book(id, name, price)
  }
```

修改查询代码

```
def query1[A](sql: String, conn: Connection)(implicit mapper: Mapper1[A]) : List[A] = {
  var stmt: Statement = null
  try {
    stmt = conn.createStatement()
    stmt.executeQuery(sql).rows.map(mapper.to)
  } catch {
    case e: SQLException =>
      List()
  } finally {
    if (stmt != null) stmt.close()
  }
}
```

查询代码3

```
import Mapper1._
val students2: List[Student] = using(conn)(query1[Student](sql1, _))
val books2: List[Book] = using(conn)(query1[Book](sql2, _))
```

在查询代码中没有使用map方法了
map方法的调用放在了query1方法中
在查询代码也没有使用to转化函数了
to转化函数的调用放在了query1方法中
实现放在对应类型类实例Mapper1[A]中

对于Student需要类型类实例Mapper1[Student]
对于Book需要类型类实例Mapper1[Book]
对于Teacher需要类型类实例Mapper1[Teacher]

对于任意类型A都需要定义对应类型类实例Mapper1[A]
问题似乎又回到了原点，能否不用针对每个类型A都定义Mapper1[A]，
或者只定义少量的Mapper1[B]，就可以转换成任意的Mapper1[A]呢？

使用Shapeless

HList(heterogeneous list)

```
scala> case class Student(id: Long, name: String, score: Int)
defined class Student

scala> case class Book(id: Long, name: String, price: Int)
defined class Book

scala> Generic[Student]
res0: shapeless.Generic[Student]{type Repr = shapeless.::[Long,shapeless.::[String,shapeless.::[Int,shapeless.HNil]]]} = anon$macro$4$1@52a5d473

scala> Generic[Book]
res1: shapeless.Generic[Book]{type Repr = shapeless.::[Long,shapeless.::[String,shapeless.::[Int,shapeless.HNil]]]} = anon$macro$8$1@65620075
```

Student: Long :: String :: Int :: HNil
Book: Long :: String :: Int :: Hnil
Student和Book都可以表述为Long :: String :: Int :: HNil

使用Shapeless

```
implicit val hListMapper: Mapper[Long :: String :: Int :: HNil] = instance { li =>
  li(0).toLong :: li(1) :: li(2).toInt :: HNil
}
```

```
val studengGeneric: List[Long :: String :: Int :: HNil] =
  using(conn)(query1[Long :: String :: Int :: HNil](sql1, _))
val students3: List[Student] = studengGeneric.map(s => Generic[Student].from(s))

val bookGeneric: List[Long :: String :: Int :: HNil] =
  using(conn)(query1[Long :: String :: Int :: HNil](sql2, _))
val books3: List[Book] = bookGeneric.map(b => Generic[Book].from(b))
```

studentGeneric和bookGeneric都是List[Long :: String :: Int :: HNil]，而List[Long :: String :: Int :: HNil]可以转化成List[Student]，也可以转化成List[Book]。
那么只需要提供一个Mapper[Long :: String :: Int :: HNil]实例，而不用分别提供Mapper[Student]和Mapper[Book]实例。

这样处理确实减少了重复代码，但是假如我们映射的对象为：
case class Teacher(id: Long, name: String, onHoliday: Boolean)
那么我们还需要提供Mapper[Long :: String :: Boolean]
还有更通用的方法吗？

使用Shapeless

```
implicit val intMapper: Mapper[Int] = instance(_.head.toInt)

implicit val longMapper: Mapper[Long] = instance(_.head.toLong)

implicit val stringMapper: Mapper[String] = instance(_.head)

implicit val booleanMapper: Mapper[Boolean] = instance(li => if (li.head == "0") false else true)

implicit val hNilMapper: Mapper[HNil] = instance(li => HNil)

implicit def hListMapper[T, H <: HList](implicit ma: Mapper[T],
                                        mb: Mapper[H]): Mapper[T :: H] = instance {
  case Nil => throw new IllegalArgumentException(s"The empty List cannot be converted to HList")
  case li: List[String] => ma.to(List(li.head)) :: mb.to(li.tail)
}
```

```
case class Teacher(id: Long, name: String, onHoliday: Boolean)
val sql3 = s"SELECT * FROM teacher"
val teacherGeneric: List[Long :: String :: Boolean :: HNil] =
  using(conn)(query1[Long :: String :: Boolean :: HNil](sql3, _))
val teachers: List[Teacher] = teacherGeneric.map(t => Generic[Teacher].from(t))
```

使用Shapeless

```
Mapper[Long :: String :: Boolean :: HNil] =>
hListMapper[Long :: String :: Boolean :: HNil] =>
hListMapper(Mapper[Long], Mapper[String :: Boolean :: HNil]) =>
hListMapper(Mapper[Long],
        hListMapper(Mapper[String],  Mapper[Boolean :: HNil])) =>
hlistMapper(Mapper[Long],
        hListMapper(Mapper[String],
                hListMapper(Mapper[Boolean], Mapper[HNil])))
```

其中:Mapper[Long]、Mapper[String]、Mapper[Boolean]、Mapper[HNil]在代码中都已经提供。

目前看来我们只需要提供基本类型T的Mapper[T]，以及Hlist类型[T, H <: HList]的Mapper[T :: H]就好了，这样代码确实更加通用了。但是我们每次都是将List[String]转化成genericT，然后再将genericT通过Generic[T]的from方法转换成T，这个地方可不可以用implicit和type class来处理的，不如试一试

使用Shapeless

```
implicit def genericMapper[A, R](implicit
                                 gen: Generic[A] { type Repr = R },
                                 mr: Mapper[R]): Mapper[A] =
  instance { li => gen.from(mr.to(li)) }
```

```
val students4: List[Student] = using(conn)(query1[Student](sql1, _))
val book4: List[Book] = using(conn)(query1[Book](sql2, _))
```

Mapper[Student] =>
genericMapper(Generic[Student], Mapper[Long :: String :: Int :: HNil])
Mapper[Student] =>
genericMapper(Generic[Book], Mapper[Long :: String :: Int :: HNil])
Mapper[Long :: String :: Int :: HNil] =>
Mapper[Long] Mapper[String] Mapper[Int] Mapper[HNil]

至此不用为专门提供无限的类型Mapper[Student]，Mapper[Book]，Mapper[Teacher]…只需要提供有限类型Mapper[Long]，Mapper[Int]，Mapper[String]，Mapper[Boolean]，Mapper[HNil]，HListMapper和GenericMapper。

总结

implicit val tTC[T]: TC[T] +
Implicit val hNilTC [HNil]: TC[HNil] +
Implicit def TC[T, H <: HList]: TC[T :: H] +
Implicit def genericTC[A, R]: TC[A]

上面的代码讲解的都是如何将List[String]转化成T，那ResetSet该如何转化成T？
遇到Option[T]又是如何处理的？
参考https://github.com/zdx1989/scala-jdbc