目录

jSqlBox用户手册

jSqlBox用户手册

简介

jSqlBox是一个全功能数据库持久层工具，它的主要特点体现在以下几个方面:
1.易学易用，基本上看了HelloWorld就可以开始使用它了。
2.功能全，架构优秀，当深入使用它时你会发现它几乎包含了数据库持久层涉及的所有功能，包括跨数据库DDL生成、主键生成、实体注解、分页、各种SQL写法（ActiveRecord/模板/链式写法/参数内嵌写法等)、ORM查询、批处理、声明式事务、分布式事务、缓存、日志、拉截器、主从、分库分表等。
3.它能在几乎所有关系数据库上使用，支持80多种数据库方言。
4.无会话(Sessionless)设计，能轻易与其它持久层工具混合使用。
如果想要快速了解jSqlBox与其它DAO工具的优缺点横向对比，请参见附录"与其它DAO工具的对比"一节。

jSqlBox的架构

jSqlBox由DbUtils、jBeanBox、jDialects、jDbPro、jTransactions再加上jSqlBox本身，一共6个模块组合而成，这些子模块都以源码内含的形式包含在项目中，以减少发布包的数量，但这些模块本身是独立的项目，各自在Maven中央库上也有独立发布版，可以单独下载使用。
DbUtils: 这是jSqlBox的内核，包装了JDBC的访问，DbUtils全称为Apache Commons DbUtils。
jDbPro: 是在DbUtils基础上的进一步包装，提供了各种不同的SQL写法，如参数内嵌式写法和SQL模板支持等。
jDialect: 用于处理所有与数据库方言相关的功能，如DDL生成、函数变换、分页、java与jdbc参数的类型转换等。
jTransactions: 用于提供Dao工具的事务功能，除了jSqlBox，它也可以用来为其它JDBC工具提供事务功能。
jBeanBox: 这是一个小巧的IOC/AOP工具，用来支持声明式事务功能。大项目也可以不用它而采用更常见但更臃肿的Spring。
jSqlBox: 这是项目的本体，除了整合以上模块，还实现了ActiveRecord、ORM查询、分库分表、分布式事务等功能。

jSqlBox的开发目是解决其它数据库持久层工具的一些问题，主要有（纯属个人见解）：

1 架构有问题，重复发明轮子。其它持久层通常试图提供一篮子解决方案，但是不分模块的开发方式使得代码纠缠成一堆乱麻，不可复用、难以维护和扩展。例如Hibernate、MyBatis、jFinal、NutzDao、BeetlSql、JdbcTemplate等工具，都面临着如何处理跨数据库开发的问题，它们的做法要么就是自己从头到尾开始开发一套方言工具，要么就是干脆不支持数据库方言，没有借签其它工具的成果，也没有考虑将自己的成果分享给其它工具。
2 过于执迷于某项技术，试图用一把锤子解决所有问题。例如Hibernate和JPA对象化设计过度，臃肿复杂(JPA共有100多个注解)。MyBatis的XML配置繁琐，没有提供CRUD方法(有插件如MyBatis-Plus等提供CRUD功能，但总归不如原生就支持的好，而且依然不能彻底摆脱XML配置）。JdbcTemplate和DbUtils过分偏重于底层SQL，不方便维护。
3 功能不全。持久层是一个综合工程，跨数据库、分页、主键生成、实体注解、ActiveRecord、动态SQL、ORM、缓存、事务、主从支持、分库分表等各方面都必须考虑，而且整合这些不同专题的内容，让它们减少互相干扰，保持架构的清晰、代码的精简是非常重要的。目前没有几个持久层工具能在所有方面都能做到令人满意。
4 不支持动态配置。从数据库表到Java对象，称为ORM，这就需要进行配置。但这些常见的持久层工具，不支持动态配置，这对于需要动态生成或修改配置的场合是个缺陷。例如Hibernate用注解或XML配置的实体Bean，在运行期很难更改外键关联关系、数据库表映射关系配置。
5 软件中存在一些反模式。例如:

- Hibernate、MyBatis、BeetlSql、Ebean、Nutz等工具中各自发明了一些利用Java方法来支持SQL重构的写法，类似下面这种:
  customer.billingAddress.city.equalTo("Auckland").status.isEqualTo(Status.NEW);开发时写起来很爽，但是SQL一复杂就很难看，破坏了可读性，不利于维护和移植。
- 很多工具模仿MyBatis，过分强调代码和SQL分离，采用基于接口代理的设计，不管业务需不需要，先创建接口和DAO层再说，造成中间垃圾层太多，影响开发效率。这一点，大家可以对比一下DaoBenchmark项目，虽然是个性能测试项目，但从源码数量上来说，jSqlBox也是远远少于其它DAO工具的，源码少即意味着开发效率高。
- 1+N陷阱，在Hibernate、MyBatis等工具中，常出现FetchType.LAZY之类设置，一不小心就会出现1+N问题，严重影响性能和失去对项目的掌控感。

jSqlBox的开发目标是尽量避免这些问题，它的主要优点有:

1 架构合理。jSqlBox将Jdbc工具、事务、数据库方言等功能分成子模块，由不同的子项目实现，每个子项目都可以脱离jSqlBox存在，甚至可供其它ORM工具使用，避免重复发明轮子。模块之间不存在双向藕合，每个模块源码尽量控制在30个类左右，以提高可维护性。 同时jSqlBox的各个子模块在开发中大量借签了其它成熟项目，例如，方言模块(jDialects)抽取了Hibernate的70种方言，事务模块(jTranscations)在提供简化版声明式事务的同时兼容Spring事务，Jdbc工具(jDbPro)基于成熟的DbUtils，链式风格和模板的引入则分别借签jFinal和BeetlSql，并且各自有改进，如新的链式语法、模板可以切换。
2 基于DbUtils这个稳固的内核不仅节约了开发成本，而且与DbUtils兼容，基于DbUtils的项目可以无缝升级到jSqlBox。
3 支持SQL写法多。从最底层的JDBC方法到参数内嵌式SQL写法、DataMapper、ActiveRecord、Tail模式都支持。
4 学习曲线平滑、可维护性好。例如jsqlbox模块下，源码只有40多个类，主要负责整合各个子模块、对象映射、分布式事务。方言、数据库访问等功能被分配到子模块中实现。模块化设计可以形成一个平滑的学习曲线。最极端情况是DbUtils使用者可以无学习直接上手jSqlBox。
5 技术创新多，如参数内嵌式写法(是jSqlBox的首创)、多行SQL文本支持、动态配置、对象关联查询、树结构查询等。
6 无会话(Sessionless)设计，是真正轻量级的持久层工具,能轻易和其它持久层工具混用或作为补丁存在。

7 是除Hibernate之外少有的支持几乎所有数据库DDL脚本生成(80种方言)的持久层工具，方便单元测试。
8 性能高，启动快速，几乎是0秒启动，这点对于需要反复启动的单元测试来说很有用。

在开发过程中，对于以下一些持久层工具提供的功能，jSqlBox决定暂不支持：

1 Sql文件映射到接口代理。个人认为SqlMapper是一个反模式，jSqlBox推荐用肮脏直接的方式在Java里拼写SQL(即jSqlBox的SQL参数内嵌写法)，多行SQL可以利用Text类存放，（甚至采用Java13版），支持IDE定位。
2 (支持/或不支持重构的)条件构造器，或用Java方法拼接来代替SQL，理由见上，属于意义不大的反模式。
3 懒加载（延迟加载)功能。实体属性的懒加载会增加DAO层的复杂性和可维护性，jSqlBox不支持懒加载。

目前jSqlBox的主要缺点:

jSqlBox是个小众项目，用户不多，在多人项目中选用jSqlBox这种小众DAO工具之前，必须要先和其它人沟通确认才可以，因为DAO工具是项目的基石，万一出现问题会对项目影响很大。
个人建议jSqlBox可以先在以下场景试用：个人项目、技术强能看懂源码、只使用了jSqlBox的部分功能如参数内嵌式SQL、作为其它DAO工具的补丁混搭使用这几种场合。

01 配置与入门

配置与入门

这一节介绍jSqlBox基本的配置和使用，更多关于配置的讲解请参见"jSqlBox配置详解"一节。

配置

在项目的pom.xml文件中加入以下内容即可：

```
<dependency>
  <groupId>com.github.drinkjava2</groupId>
  <artifactId>jsqlbox</artifactId>
  <version>5.0.15.jre8</version> <!-- 或Maven中央库最新版本-->
</dependency>
```

jSqlBox不依赖任何第三方库，它只需要一个单独的jar包就可以使用了，另外两种使用方式是: 可以直接到Maven中央库去下载jsqlbox-5.0.15.jre8.jar这个包放到项目库目录下，或是对于想要了解、修改源码的场合，可以将core\src\main\java目录下所有源码拷到项目中，也是可以直接使用的。

第一个jSqlBox示例:

在pom.xml中加入以下jSqlBox和任一个关系数据库驱动的依赖：

```
   <dependency>
    <groupId>com.github.drinkjava2</groupId>
     <artifactId>jsqlbox</artifactId>
     <version>5.0.15.jre8</version> <!-- 或最新版 -->
   </dependency>

   <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId> <!--这个示例使用H2内存数据库-->
    <version>2.1.210</version>
   </dependency>
```

在IDE中新建一个HelloWorld.java并粘贴以下源码：

```
import javax.sql.DataSource;
import org.h2.jdbcx.JdbcConnectionPool;
import static com.github.drinkjava2.jsqlbox.DB.*;
import com.github.drinkjava2.jdialects.annotation.jdia.UUID25;
import com.github.drinkjava2.jdialects.annotation.jpa.Id;
import com.github.drinkjava2.jsqlbox.ActiveEntity;
import com.github.drinkjava2.jsqlbox.DB;
import com.github.drinkjava2.jsqlbox.DbContext;

public class HelloWorld implements ActiveEntity<HelloWorld> {
  @Id
  @UUID25
  private String id;
  private String name;
  public String getId() {return id;}
  public void setId(String id) {this.id = id;}
  public String getName() {return name;}
  public HelloWorld setName(String name) {this.name = name;return this;}

    public static void main(String[] args) {
     DataSource ds = JdbcConnectionPool  //新建数据源，这个示例使用H2内存数据库
                 .create("jdbc:h2:mem:demo;MODE=MYSQL;TRACE_LEVEL_SYSTEM_OUT=0", "sa", "");
     DbContext ctx = new DbContext(ds);
     ctx.setAllowShowSQL(true);  //开启SQL日志输出
     DbContext.setGlobalDbContext(ctx);  //设定全局上下文
     ctx.quiteExecute(ctx.toDropAndCreateDDL(HelloWorld.class));  //生成DDL,建数据库表
     ctx.tx(() -> {  //开启事务
        HelloWorld h = new HelloWorld().setName("Foo").insert().putField("name", "Hello jSqlBox").update();
        System.out.println(DB.qryString("select name from HelloWorld where name like", que("H%"),
```

```
      " or name=", que("1"), " or name =", que("2")));
        h.delete();  //删除实体
      });
      ctx.executeDDL(ctx.toDropDDL(HelloWorld.class)); //删除数据库表
    }
  }
```

运行它，打印出"Hello jSqlBox", 说明在数据库中插入了一条记录。Bingo! 打完收工，没有什么接口、实现、Dao之类的垃圾代码。

例子中的事务处理 ctx.tx(()->{...}) 也可以写成传统的写法：

```
      try {
         ctx.startTrans(); //开启事务
         HelloWorld h = new HelloWorld().setName("Foo").insert() ;
         h.delete();
         ctx.commitTrans(); //提交事务
      } catch (Exception e) {
         ctx.rollbackTrans(); //回滚事务
      }
```

例子中的ctx.quiteExecute(ctx.toDropAndCreateDDL(HelloWorld.class));方法将实体Bean转化为对应数据库方言的DDL脚本并执行。

使用jSqlBox首先要创建一个DbContext实例，有以下两种构造方法：
DbContext()
DbContext(DataSource)
当构造器参数为空时，表示创建一个无数据源的DbContext实例，这种情况下因为DbContext不知道数据源在哪里，使用时必须在每个方法中添加一个Connection实例作为参数传给它，例如ctx.qryMapList(conn, "select * from users where name=", DB.que("tom"));

当构造器参数存在一个DataSource实例时，表示创建一个含数据源的DbContext实例，这种情况下方法里不需要Connection实例作为参数，例如ctx.qryMapList("select * from users where name=?", par("tom"));

对DbUtils熟悉的同学可能对以上用法看着眼熟，因为DbContext是QueryRunner的子类，所以也有与DbUtils的QueryRunner相似的构造器，并继承了QueryRunner的所有方法。所以如果对ORM不感兴趣，也可以直接将jSqlBox当作一个SQL工具来使用，利用它强大的SQL/Java混写能力提高开发效率。 上面例子中que()和par()方法是DB类中定义的静态方法，用于将Java变量值转换为SQL参数，防止SQL注入攻击，这两个方法通常以静态引入的方式使用。

复杂情况下DbContext实例的设置，例如日志、模板、方言、事务、拦截器类等，可以直接设定它的各个选项，例如ctx.setAllowShowSQL(true)将打开日志输出， 详情请参见"jSqlBox的配置"一章。

注意： DbContext实例除了配置方法不安全外(即setXxx和getXxx之类的方法)，它的其它所有SQL操作方法都是线程安全的 ，所以如果只有一个数据源，整个项目可以只创建一个DbContext实例就够用了，而且通常这一个实例还会被配置成全局静态实例，这样可利用DB类中的静态方法来静态引入使用，极大地简化了编程。不过如果想要创建很多个DbContext实例也没问题，它是一个轻量级对象，在手工设定方言(以避免每次读取数据库MetaData)的前提下，一秒内创建几万个实例也不是问题，只是通常没必要。

jSqlBox默认工作在自动提交模式。如果要使用事务，除了上面HelloWorld示例中的tx()事务模板方法外，还自带声明式事务的支持，请详见“事务配置”一节介绍，以及demo目录下的几个示范项目"jsqlbox-jfinal"、"jsqlbox-springboot"、"jsqlbox-mybatis"、"jsqlbox-actframework"等。

在Spring环境中使用

jSqlBox自带声明式事务功能，可以独立使用，但它也支持使用Spring来管理事务，比如使用SpringBoot,添加jSqlBox库依赖后，它的事务配置很简单：

```
public class MyAppDemo{
   @Autowired
   DataSource ds;

   public static void main(String[] args) {
      SpringApplication.run(MyAppDemo.class, args);
   }

   @Bean
   public DbContext createDefaultDbContext() {
     DbContext.setGlobalNextDialect(Dialect.MySQL5InnoDBDialect);//手工设定方言，以避免读取数据库MetaData
     DbContext ctx = new DbContext(ds);
     ctx.setConnectionManager(SpringTxConnectionManager.instance());//事务管理器
     DbContext.setGlobalDbContext(ctx);// 设定静态全局上下文，供DB静态方法和ActiveRecord使用
     return ctx;
```

```
    }
  }
```

注意这个配置除了可以单独使用jSqlBox外，还同时支持了jSqlBox和Hibernate或MyBatis混用在同一个事务中，因为jSqlBox的事务被托管给Spring的事务管理器SpringTxConnectionManager的实例了。这个例子位于demo目录下的jsqlbox-springboot。

02 分页、DDL和实体源码生成

jSqlBox所有与分页、DDL生成、实体源码生成、主键、函数变换、实体注解解析等与跨数据库相关的功能，都依赖于jDialects模块。
jDialects是一个独立的项目，支持80多种数据库方言，支持十多个主要的JPA注解，它可供第三方DAO工具使用，关于它的详细介绍，请移至它的主页了解：jDialects主页

当使用DbContext ctx= new DbContext (dataSource)方法创建DbContext实例后，就可以用ctx.getDialect()方法来获取当前数据源的Dialect实例。另外强烈建议手工指定DbContext的方言类型，见"jSqlBox的配置"一章，在程序启动阶段，用DbContext.setGlobalNextDialect()方法来设定好一个全局默认方言类型，这样就可以加快DbContext的创建，因为跳过了读取数据库信息并猜测方言这个过程，可以更明确地指定方言类型。

jDialects中的TableModel类在jSqlBox中占据重要地位，TableModel是一个虚拟的、与实际数据库无关的数据表模型，jSqlBox也依赖于这个虚拟表模型进行对实体动态配置的支持，请详见"固定和动态配置"一章。TableModel包含有多个ColumnModel,对应数据表列。

jDialects有许多功能，详情请见它的主页，此处只作简单介绍：
1.实体注解配置
jDialects自带的与JPA重名的注解有(JPA注解有90多个，jSqlBox只支持以下这15个)：
@Column,@Convert,@Entity,@Enumerated,@GeneratedValue,@GenerationType,@Id,@Index,@SequenceGenerator,@Table,@TableGenerator,@Temporal,@Transient,@UniqueConstraint,@Version
一般的原则是，如果jDialects的注解与JPA完全重名，则表示它的用法与JPA注解完全相同，所以可以参考JPA教材学习即可。除了标准JPA注解外，jDialects还自带一些非标的注解，请详见"实体注解配置"一节内容。

另外，用户也可以使用Dialect.setNamingConversion()方法来设定全局的命名映射规则而不需要在每个实体上添加注解，如：

```
Dialect.setGlobalNamingConvension(NamingConvension.LOWER_CASE_UNDERSCORE); //所有表名和列名将映射为小写下划线，如xx_xxx格式。
```

当setNamingConversion被设定后，也依然可以使用@Table和@Column注解来调整个别表名、列名的映射，注解具有最高优先级。

1.跨数据库的DDL脚本生成功能
String[] ddls = ctx.toCreateDDL(HelloWorldTest.class);
ctx.toCreateDDL()等同于ctx.getDialect().toCreateDDL方法，可以将实体类转化为对应当前数据库的DDL脚本。类似的还有toDropDDL方法(生成删除脚本)和toDropAndCreateDDL方法(生成删除和创建脚本)。

2.从数据库生成Java实体类源码
以下语句会读取数据库所有表格并在指定的"c:\temp"目录下生成与所有数据库表格对应的实体类源码：

```
Map<String, Object> setting = new HashMap<String, Object>();
setting.put(TableModelUtils.OPT_EXCLUDE_TABLES, Arrays.asList("Dbsample")); // 排除个别表名
setting.put(TableModelUtils.OPT_PACKAGE_NAME, "somepackage");// 包名
setting.put(TableModelUtils.OPT_IMPORTS, "import java.util.Map;\n"); // 追加新的imports
setting.put(TableModelUtils.OPT_REMOVE_DEFAULT_IMPORTS, false); // 是否去除自带的imports
setting.put(TableModelUtils.OPT_CLASS_DEFINITION, "public class $ClassName extends ActiveRecord<$ClassName> {");// 类定义模板
setting.put(TableModelUtils.OPT_FIELD_FLAGS, true); // 全局静态属性字段标记
setting.put(TableModelUtils.OPT_FIELD_FLAGS_STATIC, true); // 全局静态属性字段标记
setting.put(TableModelUtils.OPT_FIELD_FLAGS_STYLE, "upper"); // 全局静态属性字段标记可以有upper,lower,normal,camel几种格式
setting.put(TableModelUtils.OPT_FIELDS, true); // 是否生成JavaBean属性
setting.put(TableModelUtils.OPT_GETTER_SETTERS, true); // 是否生成getter setter
setting.put(TableModelUtils.OPT_PUBLIC_FIELD, false); // JavaBean属性是否定义成public
setting.put(TableModelUtils.OPT_LINK_STYLE, true); // getter/setter是否生成为链式风格

quietDropTables(DbSample.class, studentSample.class);
createTables(studentSample.class, DbSample.class);
TableModelUtils.db2JavaSrcFiles(ctx.getDataSource(), ctx.getDialect(), "c:/temp", setting);
```

db2JavaSrcFiles方法的第一个参数是数据池，第二个参数是方言，第三个参数是输出目录，第四个参数是细节设定，详见上例setting的解释。

如果只是想针对数据库生成一种记录所有列名常量的Q类，用于拼接原生SQL(详见jSqlBox中"写出可重构的SQL"一节)，可以不用进行复杂的细节配置，直接调用以下现成的方法即可：

```
TableModelUtils.db2QClassSrcFiles(ctx.getDataSource(), ctx.getDialect(), "c:/outputFolder", "com.demo", "Q");
```

3.将实体结构或数据库结构导出到Excel
以下语句分别将指定包、指定的实体类、数据库导出为Excel的CSV格式

```
TableModelUtils.entityPackage2Excel("com.abc.domain", "d:/packageOutput.csv");
```

```
TableModelUtils.entity2Excel("d:/entitiesOutput.csv", User.class, Customer.class, Order.class);
TableModelUtils.db2Excel(dataSource.getConnection(), Dialect.MySQL57Dialect, "d:/dbOutput.csv");
```

如果只想比较两个数据库结构是否有差异，可以用以下方法：
TableModelUtilsOfDb.compareDbIgnoreLength(connection1, connection2);

4.分页功能
在出现SQL文本的地方，都可以用dialect.pagin(pageNo, pageSize, sql)方法将一个普通SQL转变为与当前数据库方言对应的分页SQL,在jSqlBox中, ctx.pagin()方法等同于ctx.getDialect().pagin()方法。
分页示例：
List<Map<String, Object>> users = ctx.qry(new MapListHandler(), ctx.pagin(2, 5, "select concat(firstName, ' ', lastName) as UserName, age from users where age>?"), 50);

基于pagin方法基础上，jSqlBox还提供了使用更简洁的分页拦截器DB.pagin(pageNo,pageSize)来进行分页, 例如:
List users = DB.entityFindAll(User.class, " where age>", DB.que(0), DB.pagin(2, 5)); 拦截器等参数可以在大多数SQL方法和CRUD方法中的任意位置出现，这是jSqlBox项目的一个特点(详见iXxx等系列方法介绍)，不象其它DAO工具，jSqlBox没有专门定义的分页查询方法，而是所有的DB类的SQL方法和ActiveRecord类的CRUD方法中都可传入分页拦截器。

当不确定是否分页拦截器存在时，还可以传入禁止分页拉截器noPagin()，示例如下：

```
public void pageQuery(Object... conditions) {
 qryLongValue("select count(1) from sys_user where 1=1 ", conditions, noPagin());
 qryEntityList(DemoUser.class, "select * from sys_user where 1=1 ", conditions, " order by id");
}
调用：
pageQuery(" and age>",que(10),pagin(1,10));
```

5.函数变换功能
不同的数据库支持的函数千差万别，利用jDialects,在出现SQL文本的地方，都可以用dialect.trans(sql)方法将一个包含"通用函数"的SQL转变成对应当前数据库方言的原生SQL，在jSqlBox中, ctx.trans()方法等同于ctx.getDialect().trans()方法。

6.分页和函数变换结合
ctx.paginAndTrans(pageNumber, pageSize, sql)方法可以同时对sql进行分页和函数变换。

7.主键生成和JPA注解支持
jDialects支持十种主键生成方式及注解、一个分布式主键生成器(雪花算法), 并兼容主要JPA注解，请详见它的主页及jSqlBox的分库分表一节。
常用的是@UUID26, @UUID(等同于@UUID32), @UUID36, 分别表示不同长度由一定算法生成的唯一UUID。这些UUID注解并不能代替@Id注解，如果是主键列，必须同时还要添加@Id注解。

jDialects的开发目的是作为一个独立的工具共享出来使用，它在Maven上有单独发布包，它没有任何第三方依赖。反之，jSqlBox依赖于jDialects这个模块并用源码内嵌的方式包含在项目中。

03 传统的SQL写法

传统的SQL写法

jSqlBox的主类DbContext是继承于DbUtils的QueryRunner, 在介绍jSqlBox这道正菜之前，本节先对继承于DbUtils的方法作个简单介绍。

带Connection参数的系列方法

这是DbUtils对JDBC的包装，手工负责连接的获取和关闭，在exceute、query等方法中要给出一个Connection参数，DbContext是QueryRunner的子类，所以有了DbContext实例就可以直接使用QueryRunner的所有方法。关于QueryRunner的方法列表请详见DbUtils的使用手册。

带Connection参数的方法示例:

```
DbContextctx = new DbContext(dataSource);
for (int i = 0; i < REPEAT_TIMES; i++) {
   Connection conn = null;
   try {
      conn = ctx.prepareConnection();
      ctx.execute(conn, "insert into users (name,address) values(?,?)", "Sam", "Canada");
      ctx.execute(conn, "update users set name=?, address=?", "Tom", "China");
      Assert.assertEquals(1L, ctx.queryForObject(conn,
            "select count(*) from users where name=? and address=?", "Tom", "China"));
      ctx.execute(conn, "delete from users where name=? or address=?", "Tom", "China");
   } catch (SQLException e) {
      e.printStackTrace();
   } finally {
      try {
         ctx.close(conn);
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
}
```

如果在始初化DbContext时没有给定DataSource参数，则Connection必须要用dataSource.getConnection()等方法手工获得。
如果在始初化DbContext时给了一个DataSource实例作为参数，在运行期也可以用ctx.prepareConnection()方法来获取一个Connection实例。

DbUtils对Jdbc的包装还比较原始，这种用法必须手工捕捉SQLException异常。

无Connection参数的方法

这种用法也是继承于DbUtils, 在execute、query等方法中不放入Connection参数，使用这种用法时必须在构造DbContext实例时提供一个DataSource参数:

```
   DataSource dataSource=....;
   DbContextctx=new DbContext(dataSource);
   try {
     ctx.execute("insert into users (name,address) values(?,?)", "Sam", "Canada");
     ctx.execute("update users set name=?, address=?", "Tom", "China");
     Assert.assertEquals(1L,
         ctx.queryForObject("select count(*) from users where name=? and address=?", "Tom", "China"));
     ctx.execute("delete from users where name=? or address=?", "Tom", "China");
   } catch (SQLException e) {
     e.printStackTrace();
   }
```

这种用法程序员必须手工捕捉SQLException并处理, 根据业务要求来决定是否要重新抛出一个运行期或业务类异常。
以上两种用法可见DbUtils实际使用还是非常麻烦的，因为需要捕获和处理异常。

注意：当没有对DbContext进行事务配置时，每一个SQL方法如execute、query等都工作在自动提交模式，也就是说，各个SQL方法是互相独立的，没有被事务保护，适用于只读或不重要的场合。当要求各个方法工作在同一个事务中时，必须在创建DbContext实例时进行它的事务配置，详见“事务配置"一节。

04 参数内嵌式SQL

参数内嵌式SQL

参数内嵌式SQL是jSqlBox的首创，在SQL里直接写参数，SQL执行时自动转化为preparedStatement，这种方式的优点是被赋值的字段和实际参数可以写在同一行上,字段很多时利于维护，也方便根据不确定的条件动态拼接SQL。SQL参数必须放在par或que方法里，如果是SqlResultHandler、拉截器、Connection、DbContext, SqlItem等已知类型的对象，则不必用方法括住。字符串类型如果不放在par或que方法里的话则视为SQL文本的一部分。
很多场合业务逻辑不复杂，但是字段很多，SQL写得很长，当要添加、修改一个字段时，光是找到这个字段和它对应的是哪一个参数就很麻烦(用模板是一种方案，但模板占位符要多打几个字，模板本身的快速定位查找也是个问题，因为通常IDE不支持定位到XML或文本文件的某一行。) 利用SQL内嵌参数这种写法，可以方便地增加、删除字段，因为每一个字段和它对应的实参都写在了同一行上。
参数内嵌式SQL是jSqlBox5.0版起的默认书写格式，所有以qry/ins/exe/upd/entity打头的方法都采用参数内嵌式SQL写法。

参数内嵌式SQL示例：

```
DbContext db= new DbContext(dataSource);
db.exe("insert into users (", //
        " name ,", par("Sam"), //一个参数写一行
        notNull("phone", user.getPhone()), //如不为null则动态添加SQL片段
        when(age>0 && age<30, "age,", par(user.getAge())), //根据条件动态添加SQL片段
        " address ", par("Canada"), //
        ") ", valuesQuestions()); //根据参数个数补上 values(?,?...?)片段
db.exe("update users set name=?,address=?", par("Tom", "China"));//参数也可以连写
Assert.assertEquals(1L, ctx.qryObject("select count(*) from users where name=? and address=?", par("Tom", "China")));
db.exe("delete from users where name=", que("Tom"), " or address=", que("China"));//问号也可以省
```

上例中SQL只有几行，还看不出它的优点，但是如果有二十行、二十个参数，就能体会这种分行写法的好处了。

在DB中定义了大量的静态方法供使用，当系统中设定好一个DbContext默认全局实例后，可以直接引用DB工具类中的SQL方法,例如:

```
DbContext.setGlobalDbContext(new DbContext(someDataSource));
DB.exe("delete from users where userId=",que(1)); //直接使用静态方法
```

上述静态引入DB类的SQL执行方法的局限是只限于单数据源场合。

DB不是jSqlBox的核心类 ，如果对它的方法命名不满意，用户可以根据它的源码写出自己喜欢的静态方法库，以方便静态引入。

上例的nutNull、par、que等方法是用“import static com.github.drinkjava2.jsqlbox.DB.*; ”这种静态引入方式使用。que(或ques)与par(或param)方法的区别是que会在原地留下一个问号字符串去拼SQL，而par仅会返回一个“”空字符串，到底用que还是par要视SQL具体情况而定，见下面：

```
ctx.exe("insert into users (", //
  " name", par(name), //
    when(age!=null," ,age ", par(age)), //when是条件判断，相当于if
  " ,address ", par(address), //注意这里只能用par，因为这个位置SQL不能出现问号
  ") ", valuesQuestions()); //根据参数个数生成values(?,?...?)片段, valuesQuestions()也可简写成VQ
ctx.exe("update users set ", //
  " name=", que(name), //这里也可以写成 " name=? ", par(name)
  notNull(", age=", age), //仅当age不为null时才插入这行
  when(address!=null, ", address=",  que(address)), //
  " where name is not null"
);
Assert.assertEquals(1L, ctx.qryLongValue(//
  "select count(*) from users where 1=1 ", //
  notBlank(" and name=", name), //仅当name不为空时才插入这行
                noBlank(" and name like","%",name,"%"); //模糊查询，仅当name不为空时才插入这行
  when("Tom".equals(name)," and name=", que(name)),//
  when("China".equals(address)," and address=", que(address)),//
  " order by name"
));
```

这有点类似模板语言，但比模强的地方是无需学习模板语法，Java本身就是最好的模板，而且Java方法可以随时自定义添加,具体怎么添加大家可以看一下DB.par()、DB.when()等方法的源码就明白了，仅有1行代码。当只做非null或非空判断时，when也可以用notNull或notBlank方法代替，后者的的最后一个参数是SQL参数，且无需用par括住。

参数内嵌式SQL所有的方法最后一个参数是一个不定长对象数组,传入的内容分为以下各种情况：

- 用来拼接SQL和参数的元素, 如：
  字符串类型: 会被解释为SQL文本
  param或par(参数1,参数2...) 会被解释为SQL参数并返回一个空字符串""，它定义在DB类中，通常静态引入使用，下同
  ques或que(参数1,参数2...) 会被解释为SQL参数, 并返回一个问号字符串"?"
  包含任意内容的数组: 会被递归解析，直到数组不再有嵌套数组为止
  notNull(sql,obj...) 最后一个参数为null时，sql和参数都不会参与拼写sql noNull(sql,obj...) 没有一个obj为null时，将所有obj参数相连成一个SQL参数,并在SQL中添加str作为SQL文本, 示例：noNull("name like ?", "%",name,"%");
  notBlank(sql,obj...) 最后一个参数为null或空字符串时，sql和参数都不会参与拼写sql noBlank(sql,obj...) 没有一个obj为null或空字符串时，将所有obj参数相连成一个SQL参数,并在SQL中添加str作为SQL文本,常用于拼接like查询
  valuesQuestions() 自动根据参数个数，生成一个values(?,?...?)SQL文本片段。也可简写成QV; CustomizedSqlItem： 自定义的特殊SQL条目，用户可以自己定义如何来翻译成SQL或参数
  when(boolean, obj...) 根据条件返回对象数组，如条件不满足则返回一个""空串, when支持嵌套，但是SQL参数必须用par或que括住，而notNull之类判空方法不需要
- 特殊类型, 如：
  pagin(pageNumber,pageSize) 会被解释为一个分页拦截器, 详见分页一节
  other(obj...)方法，将一些额外参数(通常是字段别名或显示宽度等)保存在线程局部变量，并返回一个空字符串，用DB.getOthers()可以获取保存的参数
  shardTB(shardvalues) 生成分表后的表名字符串，详见分库分表一节
  shardDB(shardvalues) 会解释为分库后的DbContext,详见分库分表一节
  shard(shardvalues) 会同时解释为分库后的DbContext和表名,详见分库分表一节
  Xxxx.class： 如果一个参数是User.class这种类型，表示SQL方法将根据User类来翻译成SQL,常用于Text类多行文本解析和实体类查询
  TableModel实例：传入一个TableModel可以进行覆盖实体到数据表的缺省配置，详见动态配置一节
  SqlResultHandler实例: 某些方法需要传入一个SqlResultHander参数，详见DbUtils
  SqlHander拉截器实例： 传入SqlHandler拦截器，详见拦截器一章
  Connection实例：传入Connection实例, 运行期由这个Connection去执行SQL
  DbContext实例：传入DbContext实例, 运行期由这个实例去执行SQL
  SqlTemplateEngine实例：当接收到一个SqlTemplateEngine接口的实例后（如DB.TEMPLATE)，SQL转为模板方式运行
  IGNORE_NULL 这是一个开关参数，当实体插入和修改时（即entityInsert/entityUpdate方法)，忽略掉实体的所有null值字段
  IGNORE_EMPTY 这是一个开关参数，当实体插入和修改时，忽略掉实体的所有null值字段和空字符串字段

对于jSqlBox的SQL方法变长参数的理解，可以将它看成是Windows操作系统下的消息，每一个SQL条目，只是一个消息而已，jSqlBox将会汇总所有消息并把它们翻译成实际的SQL或参数并执行，在jSqlBox中所有内容都可以作为参数传递，如拦截器、模板引擎、甚至DbContext实例本身，也可以作为参数传递（这种情况下，传入的DbContext实例将夺取SQL执行权，常用于多数据源场合）。另一方面，jSqlBox的大多数SQL方法、CURD方法（包括ActiveRecord的方法），都允许额外附加不限数量的SQL条目，以实现最大的灵活性，这就是为什么jSqlBox中的大多数方法最后一个参数都是一个可变对象数组参数的原因。

最后再上几个复杂点的例子，显示参数内嵌式SQL的灵活和强大：

```
//写出支持重构的SQL:
QUser u=QUser.user; //Q类详见可重构的SQL一节
ctx.exe("insert into ", u, " ( ",
 u.name, ",", par("Sam"), //
 u.address, " ", par("Canada"), //
 ") ", valuesQuesions());

//传入一个模板对象实例DB.TEMPLATE，就可以使用SQL模板了
UserAR sam = new UserAR("Sam", "Canada");
UserAR tom = new UserAR("Tom", "China");
paramMap.put("user", sam);
ctx2.exe(DB.TEMPLATE,  "insert into users (name, address) values(#{user.name},:user.address)", paramMap);
ctx2.exe(DB.TEMPLATE,"update users set name=#{user.name}, address=:user.address", bind("user", tom));
Assert.assertEquals(1L,
        ctx2.qryLongValue(TEMPLATE,"select count(*) from users where name=#{name} and address=:addr",
        bind("name", "Tom", "addr", "China")));

//other方法可以存放任意额外信息，在当前线程用DB.others()方法可获取这些额外信息
Map<String, Object> result = DB.qryMap("select ", //
  " id", other("id", 10), //
  when(u.age>10, ", name as child_name ", other("姓名", "显示列宽=20", "注：用红字显示")), //
  when(u.age<10, ", name as adult_name ", other("姓名", "显示列宽=10")), //
  " from TitleDemoEntity", //
  " where id<>", que("a"), //
  when(name != null, " and name like ", que("%" + name + "%")), //
  new PrintSqlHandler() //
);

//万物皆可传
```

```
new User(100, "Tom", "China").update(ctx2," and age>?", param(5), IGNORE_EMPTY, new PrintSqlHander());
```

上例最后一行做了以下事情：
手工切换到ctx2这个DbContext实例上(即操作另一个数据源)
ActiveRecord实体User主键为100的记录，如果age字段大于5则更新它的name为"Tom",address为"China" 勿略User实体的所有null或空值属性
打印SQL到控制台

DbContext和DB类中定义的参数内嵌式SQL方法一览:

```
qry(Object...) 执行一个查询，返回类型由传入的SqlResultHander或SqlHander来决定
qryObject(Object...) 执行一个查询，将第一行第一列作为Object返回，如不存在则返回null
qryLongValue(Object...) 执行一个查询，返回一个long值，如果结果不能被转为long值，则抛出运行期异常
qryIntValue(Object...) 执行一个查询，返回一个int值，如果结果不能被转为int值，则抛出运行期异常
qryBooleanValue(Object...) 执行一个查询，返回一个boolean值,如果结果不能被转为booolean值，则抛出运行期异常
qryString(Object...) 执行一个查询，返回一个字符串值
qryMapList(Object...) 执行一个查询，返回一个List<map<String,Object>>类型
qryMap(Object...) 执行一个查询，将第一行记录返回一个map<String,Object>类型
qryList(Object...) 执行一个查询，将第一列记录返回一个List<Object>类型
qryEntityList(Object...) 执行一个查询，第一个参数通常是实体类类型，返回一个实体列表
upd(Object...) 执行一个SQL, 等效于DbUtils的update方法,但不用捕获导常
ins(Object...) 执行一个insert SQL，等效于DbUtils的insert方法,但不用捕获导常
exe(Object...) 执行一个SQL, 等效于DbUtils的execute方法,但不用捕获导常
entityXxxx(Object...) 实体相关的一系列CURD方法，详见entity方法一节
```

从5.0版起，jSqlBox删除了p、i、n、t、e开头的方法，只使用参数内嵌式风格为唯一书写SQL方式，e开头的方法改为entityXxxx形式，这样改进后可读性和可维护性更好。

05 entity系列方法

entity系列方法(DataMapper模式)

本节及下一节“ActiveRecord模式”介绍对利用实体类对数据库进行CRUD操作，这种做法的优点是SQL由系统自动生成，开发效率高，实体支持重构、可维护性好。而且jSqlBox自带的分布式事务功能也只支持基于实体类的CRUD方法。

对实体进行CRUD操作，常见的有两种方式，一种方式是类似于Hibernate中的session.save(user)这种写法，也被称为Data Mapper模式，它的特点是实体类本身不带CRUD方法，而是由一个工具实例来对这个实体进行CRUD操作。另一种方式是类似user.update()这种写法，它的特点是实体类自带CRUD方法。jSqlBox同时支持Data Mapper和ActiveRecord两种模式。本节介绍的是Data Mapper模式的CRUD方法。

从易用性方面考虑，jSqlBox建议实体类继承于ActiveRecord类或实现ActiveEntity接口，但是有时候实体类因为某些特殊原因(例如同时使用多个数据源)，不能采用ActiveRecord模式，这时可以采用Data Mapper模式，这种模式的所有方法以小写字母entity打头，表示它对单个实体(Entity)进行操作，示例如下：

```
DbContext ctx=new DbContext(dataSource);
User user = new User("Sam", "Canada");
ctx.entityInsert(user);
user.setAddress("China");
ctx.entityUpdate(user);
User user2 = ctx.entityLoadById(User.class, "Sam");
ctx.entityDelete(user2);
```

ctx.entityInsert(user)方法会根据User类的实体注解的主键生成策略，生成对应的主键值（详见jDialects项目主页，共有十种主键策略，另外还有一个分布式主键类型，见分库分表一节）。

DbContext是无会话(Sessionless)的轻量级对象，实体Bean总是无状态的，不分什么VO、PO之类的，从View层传来的Form Bean可以直接用entityInsert方法存到数据库，从数据库取出来的实体bean可以直接传递到view层。
在Hibernate/JPA/MyBatis等工具中，有时一个POJO被加载后变成了一个代理实例，也就是说执行pojo.getClass().equal(POJO.class)的结果是false, 这在某些情况下会让人抓狂。 另一个代理实例的场景是执行user.getOrder()之类的方法，Hibernate/JPA/MyBatis可能在后台发起一系列SQL调用，加载Order实例，Order类又可能加载其它实体类...,这个魔术般的功能可能是某些人的心头好，但也可能是某些人的噩梦，因为实体类一多，他们就会搞不清某个简单的get方法调用后是否会在后台引来一场SQL风暴，或是不清楚懒关联的实体到底有没有被加载。
在jSqlBox中不存在这种情况，它不玩魔术，一个POJO实例总是POJO实例本身，不是一个代理。也不会在调用get/set方法后自动在后台加载其它对象玩连连看。(jSqlBox有实体关联查询这个功能，但是有专门的findRelated系列方法，而不是用实体代理类来实现，详见实体关联查询一节。)

使用Data Mapper模式，即相当于调用DbContext实例的entity系列方法，见以下列表。

DbContext与实体CRUD相关的方法,都以“entity”打头

```
public <T> T entityInsert(T entity, Object... sqlItems) 插入实体到数据库，如出错则抛出SqlBoxException运行期异常(以下简称异常)，sqlItems是可选Sql条
public <T> T entityUpdate(Object entity, Object... sqlItems) 根据实体ID更新数据库记录，如出错则抛异常
public int entityUpdateTry(Object entity, Object... sqlItems) 尝试根据实体ID更新记录，返回影响行数
public void entityDelete(Object entity, Object... sqlItems) 根据实体ID删除记录，如出错则抛异常
public int entityDeleteTry(Object entity, Object... sqlItems) 尝试根据实体ID删除记录，返回影响行数
public void entityDeleteById(Class<?> entityClass, Object id, Object... sqlItems) 根据给定ID删除记录，如出错则抛异常
public int entityDeleteByIdTry(Class<?> entityClass, Object id, Object... sqlItems) 尝试根据给定ID删除记录，返回影响行数
public boolean entityExist(Object entity, Object... sqlItems) 根据实体ID检查是否记录存在
public boolean entityExistById(Class<?> entityClass, Object id, Object... sqlItems) 根据给定ID检查是否实体记录存在
public int entityCount(Class<?> entityClass, Object... sqlItems) 统计指定实体类的所有计录数
public <T> T entityLoad(T entity, Object... sqlItems) 根据实体ID加载，如出错则抛异常
public int entityLoadTry(Object entity, Object... sqlItems) 尝试根据实体ID加载，返回满足查询条件的记录行数
public <T> T entityLoadBySQL(Class<T> entityClass, Object... sqlItems) 根据SQL加载 实体，如找不到或找到多个则抛异常
public <T> T entityLoadById(Class<T> entityClass, Object entityId, Object... sqlItems) 根据给定ID加载，如出错则抛异常
public <T> T entityLoadByIdTry(Class<T> entityClass, Object entityId, Object... sqlItems) 尝试按给定ID加载，找不到返回Null
public <T> List<T> entityFindAll(Class<T> entityClass, Object... sqlItems) 加载指定实体类List，如未找到返回空List(下同)
public <T> List<T> entityFindByIds(Class<T> entityClass, Iterable<?> ids, Object... sqlItems) 按给定ID集合加载实体类
public <T> List<T> entityFindBySQL(Object... sqlItems) 按SQL加载实体类List，如找不到返回空List集合
public <T> T entityFindOneBySQL(Object... sqlItems) 按SQL加载实体类，如果找不到返回null,找到多个返回第一个实体
public <T> List<T> entityFindBySample(Object sampleBean, Object... sqlItems) 按给定Sample查询，见“按实体样板查询”一节
public <E> E entityFindRelatedOne(Object entity, Object... sqlItems) 对指定的实体，查出与它关联的一个对象
public <E> List<E> entityFindRelatedList(Object entityOrIterable, Object... sqlItems) 对实体或集合，查出关联的对象List
public <E> Set<E> entityFindRelatedSet(Object entity, Object... sqlItems) 对实体或集合，查出关联的对象Set
public <E> Map<Object, E> entityFindRelatedMap(Object entity, Object... sqlItems) 对实体或集合，查出关联的对象Map
```

以上方法大多比较简单，不需要多解释，它们的用法实例在CrudTest.java单元测试中可以看到。

其中有一点要提的是，上面一些方法中的ID指的是实体ID，可以允当实体ID的对象有：Java值对象(如Long、String等)、JavaBean对象、MAP<String,Object>实例三种类型。
例如，如果一个实体类User有唯一主键id，类型为字符串，则它的ID可以为以下三种方式的任一种：一个ID属性值不为空的User实例、一个普通字符串、一个值为Map<"id名","ID值">的Map实例。 jsqlBox是支持多主键实体映射的。单个值对象只能用于单主键实体，Map则通常仅用于复合主键。

最后4个方法与实体关联查询有关，详见"实体关联查询"一节介绍。

06 ActiveRecord模式

ActiveRecord模式通常用于单数据源场合。

继承于ActiveRecord类或实现了ActiveEntity接口的实体类(Java8或以上版本)，将拥有insert/update/delete等CRUD方法。ActiveRecord模式的优点是基于实体类操作数据库，支持实体的重构，开发效率高、可维护性好。但是从原理上来说，它与Data mapper模式中的CRUD方法实现的功能是等效的，并且在底层共用同一套函数库。如果查看ActiveRecord或ActiveEntity的源码就明白了，源码只有约300行，而且大多数内容都是在调用DbContext的方法，相当于一个中转站而已。

以下为ActiveRecord模式的一个示例：

```
public class User extends ActiveRecord<User> {
  ...
}

User user = new User();
user.setName("Sam");
user.setAddress("Canada");
user.insert();
user.setAddress("China");
user.update();
User user2 = user.load("Sam");
user2.delete();
```

使用上述ActiveRecord方式时，要注意在程序启动时调用静态方法DbContext.setGlobalDbContext(ctx)方法设定一个全局缺省上下文。

对于多数据源场合，不能只使用一个全局上下文了，所以在多数据源场合，就必须采用Data Mapper模式，或将上下文作为参数传递:

```
dbContext2.entityInsert(user); //Data Mapper模式
user.insert(dbContext2); //手工指定上下文
```

或是配置实体的@ShardDatabase属性(见分库分表一节)。

对于特殊场合，实体不能继承于ActiveRecord类（Java的单继承被占用)，如果使用了jsqlbox的jre8版本很容易解决，只需要实现了ActiveEntity接口，不需要实做任何方法，就自动拥有了CURD方法。

其它ActiveRecord工具的CRUD方法通常不带参数，而在jSqlBox中，ActiveRecord的CRUD方法中允许附加不限个数的各种SQL条目（称为SqlItem），这是jSqlBox的一个特点，例如下面这个ActiveRecord的update语句：

```
new User(100, "Tom", "China").update(ctx2," and age>?", param(5), new PrintSqlHander()， IGNORE_EMPTY);
```

做了以下事情：

1. 切换到ctx2这个DbContext实例上(即操作另一个数据源)
2. 主键userId=100的记录，如果age字段大于5则更新它的内容，且勿略实体的所有null或空值属性
3. 打印SQL到控制台
   另外，上例中实体id字段上如果有@ShardTable或@ShardDatabase注解，则上述操作中，这个实体会根据id的值自动选取分库或分表，详见分库分表一节。

ActiveRecord并不是jSqlBox中的关键类 ，正如开头所说，它只是一个中转站，jSqlBox内核并不依赖于ActiveRecord类，所以用户如果对jSqlBox自带的ActiveRecord或ActiveEntity接口中的方法命名不满意(例如与用户的实体方法名有冲突），可以参照ActiveRecord的源码，写出自己想要的ActiveRecord类。

附：ActiveRecord类的主要方法

```
public DbContext ctx(Object... sqlItems) 获取当前实体基于的DbContext实例
public T insert(Object... sqlItems) 插入实体到数据库，如出错则抛出SqlBoxException运行期异常(以下简称异常)
public T update(Object... sqlItems) 根据实体ID更新数据库记录，如出错则抛异常
public int updateTry(Object... sqlItems) 尝试根据实体ID更新记录，返回影响行数
public void delete(Object... sqlItems) 根据实体ID删除记录，如出错则抛异常
public int deleteTry(Object... sqlItems) 尝试根据实体ID删除记录，返回影响行数
public void deleteById(Object id, Object... sqlItems) 根据给定ID删除记录，如出错则抛异常
public int deleteByIdTry(Object id, Object... sqlItems) 尝试根据给定ID删除记录，返回影响行数
public boolean exist(Object... sqlItems) 根据实体ID检查是否记录存在
```

public boolean existById(Object id, Object... sqlItems) 根据给定ID检查是否实体记录存在
public int countAll(Object... sqlItems) 统计所有同类实体的所有计录数
public T load(Object... sqlItems) 根据实体ID加载，如出错则抛异常
public int loadTry(Object... sqlItems) 尝试根据实体ID加载，返回满足查询条件的记录行数
public T loadById(Object id, Object... sqlItems) 根据给定ID加载，如出错则抛异常
public T loadByIdTry(Object id, Object... sqlItems) 尝试按给定ID加载，返回行数
public List<T> findAll(Object... sqlItems) 加载所有同类实体List，如未找到返回空List(下同)
public List<T> findByIds(Iterable<?> ids, Object... sqlItems) 按给定ID集合加载实体类
public List<T> findBySQL(Object... sqlItems) 按SQL加载实体类List
public List<T> findBySample(Object sampleBean, Object... sqlItems) 按给定实体的Sample加载实体类List(见下节)
public <E> E findRelatedOne(Object... sqlItems) 对当前实体，查出与它关联的一个对象,详见“对象关联查询”一节（下同）
public <E> List<E> findRelatedList(Object... sqlItems) 对当前实体，查出关联的对象List
public <E> Set<E> findRelatedSet(Object... sqlItems) 对当前实体，查出关联的对象Set
public <E> Map<Object, E> findRelatedMap(Object... sqlItems) 对当前实体，查出关联的对象Map
getField(String) 获取指定实体属性的值，详见Tail和混合模型一节
getTail(String) 获取指定Tail列的值，详见Tail和混合模型一节
putTail(Object...) 设定Tail列的值，详见Tail和混合模型一节
putField(Object...) 设定实体属性的值，详见Tail和混合模型一节
forFields(String...) 准备对一批实体属性进行赋值，详见Tail和混合模型一节
forTails(String...) 准备对一批Tail属性进行赋值，详见Tail和混合模型一节
putValues(Object...) 对一批实体或Tail属性进行赋值，详见Tail和混合模型一节
public String shardTB(Object... sqlItems) 生成一个ShardTB SQL条目，详见分库分表一节
public DbContext shardDB(Object... sqlItems) 生成一个ShardDB SQL条目，详见分库分表一节
public Object[] shard(Object... items) 同时生成一个ShardTB和ShardDB条目，详见分库分表一节

07 Tail和混合模型

Tail和混合模型

它是ActiveRecord类特有的按实体属性名或Tail列名赋值/取值的方法，因为写法比较特殊，所以单独列出来，这两种写法因为不支持重构，破坏了实体ORM的这个重要优点，所以不建议在项目中大量使用。

基于实体属性的赋值/取值方法

1.基于实体属性的赋值

```
//基于实体属性的链式赋值之一
new User().putField("id","u1").putField("userName","user1").insert();

//基于实体属性的链式赋值之二
new Address().putField("id","a1","address","address1","userId","u1").insert();

//基于实体属性的批量赋值
new Email().forFields("id","emailName","userId");
new Email().putValues("e1","email1","u1").insert();
new Email().putValues("e2","email2","u1").insert();
```

对于ActiveRecord实体来说，建议setter也设成链式风格。jDialect也支持自动生成链式风格的setter实体类源码，详见“分页、DDL、源码生成”一节。
基于实体属性的赋值/取值(getField)可以用普通的setter/getter代替,实际上它在后台也是调用了setter/getter方法。它唯一的亮点就是可以将赋值写成一行，但代价是失去了重构支持，所以要少用。

基于Tail的赋值/取值方法

它不需要实体类中有对应的字段名，所以又称为Tail模式。
对于e系列和ActiveRecord的insert/update/delete方法，必须加一个tail(表名)指明当前操作的数据库表名，所有tail字段将与实体字段混合在一起参与CRUD操作, 否则将忽略所有tail字段，如果表名为空，表示使用当前实体所在的表。如果一个tail字段在数据库中不存在对应的列，更新或读取时将不参与生成SQL。

jsqlBox的ActiveRecord是一个混合模型，工作于两种模式：“按实体字段名” 和 “按实体字段名+tail字段名混合模式”，区别就在是否有一个tail方法出现在CURD方法中。

```
//基于tail的链式赋值之一 (tail方法里参数为空，表示使用与实体相同的数据库表)：
new User().putTail("id","u1").putTail("user_name","user1").insert(DB.tail());

//基于tail的链式赋值之二 (tail方法静态引入使用)
new Address().putTail("id","a1","usr_address","address1","user_id","u1").insert(tail());

//基于tail的批量赋值
user.forTails("user_name", "age", "birth_Day");
user.putValues("Foo", 30, new Date()).insert(tail("user_tb"));
user.putValues("Bar", 40, new Date()).insert(tail("user_tb"));
```

tail赋值方法和其它链式方法可以混用：

new User().setId("u1").putTail("usr_addr","Beijing").putField("userName","Tom").update(tail());

对于ActiveRecord类，查询时未映射到实体字段名的列会自动添加成tail，不需要tail参数：

User u = entityLoadBySQL(User.class, "select *, 'China' as user_addr from tail_demo");
System.out.println(u.getTail("user_addr")); //user_addr没有对应的实体字段，是一个tail列

对于多张表参与的实体关联查询，如果实体是ActiveRecord或TailType实例，也支持Tail模式

tail列在实体关联查询时要加上"alis_"前缀以区分是哪个实体的tail：

```
EntityNet net = ctx.qry(new EntityNetHandler(), User.class, UserRole.class, Role.class, giveBoth("r", "u"), //
            "select u.**, u.username as u_name2, ur.**, r.**, r.id as r_id2 from usertb u ", //
            " left join userroletb ur on u.id=ur.userid ", //
            " left join roletb r on ur.rid=r.id ");
```

```
        Set<User> userList = net.pickEntitySet("u");
        for (User u : userList)
            System.out.println(", name2:" + u.getTail("name2"));
```

当不想定义实体类时，可以直接使用一个Tail类来操作数据库

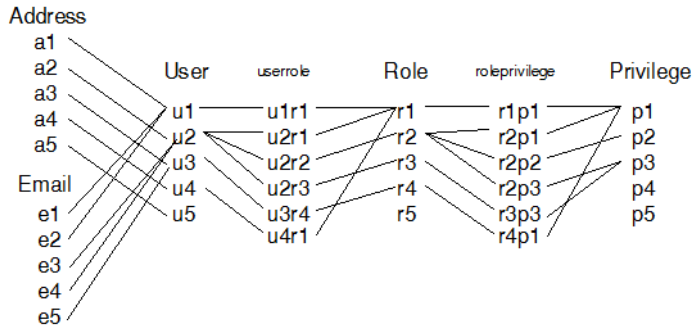
Tail类是ActiveRecord的子类，源码只有两行，它相当于实体类的字段名为空的特例：
net Tail().putTail("id","id1","u_name","Foo"） .insert(tail("tb1").load(tail("tb2")).delete(tail("tb3"));

08-1 实体关联查询

注：jSqlBox的实体关联查询是利用SQL的join连表查询功能，对使用者的SQL能力要求较高。推存采用另外一种更简单的查询关联表的方法，请详见“08-2 jSqlBox的主从表查询”

实体关联查询

jSqlBox项目主体部分是一个依赖于jDbPro和jDialcets项目的小型模块(真正属于jSqlBox模块的源码只有40多个类)，它的主要功能就是实体CRUD操作和实体关联查询，所以也可以称jSqlBox为ORM工具。当两个或以上实体类存在互相关联的对象网状结构时，可以利用jSqlBox的实体关联查询功能来查询并加载这些对象到内存中。相比其它ORM工具繁琐的配置，jSqlBox在方法调用的同时即完成ORM查询相关的配置，SQL还可以自动生成，这两个特点结合可以极大地提高查询效率和可维护性。



示例一，先感受一下jSqlBox强悍的实体关联查询：

```
private static final Object[] targets = new Object[] { new EntityNetHandler(), User.class, UserRole.class,
        Role.class, RolePrivilege.class, Privilege.class, giveBoth("r", "u"), giveBoth("p", "u") };

public void demo() {
  EntityNet net = ctx.qry(targets, AUTO_SQL);
  List<User> userList = net.pickEntityList("u");
     for (User u : userList) {
        System.out.println("User:" + u.getId());
        Set<Privilege> privileges = u.getPrivilegeSet();
        if (privileges != null)
           for (Privilege privilege : privileges)
              System.out.println("  Privilege:" + privilege.getId());
     }
   }
//输出结果为：
User:u1
  Privilege:p1
User:u2
  Privilege:p1
  Privilege:p2
  Privilege:p3
User:u3
  Privilege:p1
User:u4
  Privilege:p1
User:u5
```

可以看到，jSqlBox的对象关联查询支持自动越级关联，它的原理是怎样的呢？下面详细解说：

示例二 手工输入SQL进行对象关联查询：

```
@Test
public void testAutoAlias() {
     insertDemoData();
     EntityNet net = ctx.qry(targets, "select u.**, ur.**, r.**, p.**, rp.** from usertb u ", //
           " left join userroletb ur on u.id=ur.userid ", //
           " left join roletb r on ur.rid=r.id ", //
           " left join roleprivilegetb rp on rp.rid=r.id ", //
           " left join privilegetb p on p.id=rp.pid ");
```

```
        List<User> userList = net.pickEntityList("u");
        for (User u : userList) {
            System.out.println("User:" + u.getId());
            Set<Privilege> privileges = u.getPrivilegeSet();
            if (privileges != null)
                for (Privilege privilege : privileges)
                    System.out.println("  Privilege:" + privilege.getId());
        }
    }
```

这个示例和示例一完成的功能是一样的，只不过将AUTO_SQL改成了实际的SQL文本，所以可以看到，AUTO_SQL (在JSQLBOX中定义并静态引入）这个Sql条目就等同于让系统自动按实体类顺序产生一个跨越多张表的Left join SQL。并且抽取实体类的大写字母的小写(很绕口)作为对应SQL表的别名，如UserRole.class的别名就是"ur"。join连接的条件是根据实体类的主键、外键的Annotation配置自动生成（见jDialects中ID和FKey注解)。所以如果实体的外键没有配置的话，是不能运用AUTO_SQL这种方式自动生成SQL而必须手工敲入SQL。

示例中give("r","u")方法表示将别名"r"对应的实体赋给"u"对应的实体的相应属性字段，字段名必须与类名同名(不分大小写)，以及一些后跟List/Set/Map后缀的集合类属性，如上例中User类中的roleList属性字段将自动被赋值。give表示将前一个对象赋给后一个对象的属性。giveBoth方法则是双向赋值。当实体类属性的命名不规则时，则不能简单地用give(A, B)这种用法，而必须明确指出被赋值实体类的属性名，如give("r","u","allRoleLists")，give方法只支持属性为实体类、Set、List、Map<Object,Object>四种类型，其中Map中的Key为实体ID,即实体的主键值,如果是复合主键将会被转换成一个字符串值。

基于SQL的对象关联查询，顾名思义，是基于SQL的，也就是说如果查询结果在同一行，则系统强行认为对应的实体存在关联关系，所以这种方式与SQL的写法密切相关。上例中的u.**是jSqlBox独有的双星写法, 等同于u.id as u_id, u.user_name as u_user_name..., 手工输入这些字段也是可以的，但显然双星写法更简洁。另外u.##这种写法也有它的意义，表示只加载所有主键、外键字段，以提高性能。

EntityNetHandler拉截器生成的结果是一个EntityNet对象，它是一个内存中的网状结构的实体对象集合，可以用pickEntityList、pickEntityMap、pickEntitySet、pickOneEntity等方法，根据别名或主键值，从这个网状结构中选出所需要返回的实体对象或对象的集合,以下分别是几种筛选示例：

```
List<User> userList = net.pickEntityList("u");//根据别名筛选成List
Set<Privilege> privileges = net.pickEntitySet(Privilege.class);//根据类名筛选成Set
Map<Object, Object> roleMap = net.pickEntityMap("r");//根据别名筛选成Map
Role r1 = net.pickOneEntity(Role.class, "r1"); //根据类名和主键值筛选单个实体
Role temp = new Role();
temp.put("id", "r4");
Role r4 = net.pickOneEntity("r", temp); //根据别名和模板对象筛选单个实体
Map<String, Object> mp = new HashMap<String, Object>();
mp.put("id", "r2");
Role r2 = net.pickOneEntity("r", mp); //根据别名和Map主键(通常是复合主键)筛选出单个实体
```

注意：拔出萝卜带出泥，筛选出的实体对象的属性中可能引用了其它一串相关的对象，这是一个对象关联网，只有当这个网中的所有对象都没有引用时，才会被垃圾管理器回收。
EntityNetHandler拦截器强制必须每个表使用别名。当只针对一个表查询时，建议使用更简单的EntityListHandler或实体CRUD方法如entityLoad、entityFind、entityFindBySQL等，这些方法不要求别名。
注：本节中的所有示例源码详见单元测试下的EntityNetTest.java。

示例三 手工指定别名:
下面这个示例分别指定了User、UserRole、Role、Privilege对应的别名为"t"、 "tr"、"r", "pr"。

```
EntityNet net = ctx.qry(new EntityNet(), User.class, UserRole.class, Role.class, alias("t", "tr", "r"),
    RolePrivilege.class, Privilege.class,alias("pr"), giveBoth("r", "t"), giveBoth("pr", "t"), AUTO_SQL, //
    " order by t.id, tr.id, r.id, rp.id, pr.id");
```

alias方法(在JSQLBOX中定义并静态引入)表示指定倒数前几个出现的实体类的别名，允许出现多个aliais方法在一个SQL里。

示例四 能不能再简化？
从第一个例子看，jSqlBox的对象关联查询已经做到非常自动化了，能不能再简单一点？可以的，第一个例子还可以这样写：

```
List<User> users = ctx.autoNet(User.class, UserRole.class, Role.class, RolePrivilege.class, Privilege.class)
            .pickEntityList(User.class);
```

极简的前题是它建立在约定的基础上，UserRole中要设定User的外键和Role的外键..., 如果User类中有Privilege相关的属性，必须命名为Privilege privilege； 或 List privilegeList； 或 Set privilegeSet；反之亦然。否则就必须用give方法手工指定属性名而不能使用autoNet方法了。另外所有的自动赋值只是针对于第一个类，例如Role和Privilege之间就不会建立关联。

示例五 分步查询:

上面几个例子都是一个SQL Left join连到底，如果对象关系太复杂，一个SQL写不出来怎么办？没关系，可以分几次来查，见下例：

```
    @Test
    public void testJoinQuary() {
        EntityNet net = ctx.qry(targets, AUTO_SQL);
        ctx.qry(net, User.class, Email.class, give("e", "u"), AUTO_SQL);
        ctx.qry(net, User.class, Address.class, giveBoth("a", "u"), AUTO_SQL);
        User u = net.pickOneEntity(User.class, "u2");

        List<Role> roles = u.getRoleList();
        if (roles != null)
            for (Role r : roles)
                System.out.println("  Roles:" + r.getId());

        if (u.getEmailList() != null)
            for (Email e : u.getEmailList())
                System.out.println("  Email:" + e.getId());

        if (u.getAddress() != null) {
            System.out.println("  Address:" + u.getAddress().getId());
            Assert.assertTrue(u == u.getAddress().getUser());
        }
    }
```

先执行一个查询，获得一个EntityNet对象，然后将这个EntityNet对象作为参数扔到SQL查询方法里，后续的查询结果就会与先前的查询结果累加起来。上例的执行结果为：

```
User:u2
  Roles:r1
  Roles:r2
  Roles:r3
  Email:e3
  Email:e4
  Address:a2
```

示例六 手工加载
先上代码再解释：

```
    @Test
    public void testManualLoad() {
        insertDemoData();
        List<User> users = ctx.eFindAll(User.class);

        for (User u : users) {
            System.out.println("User:" + u.getId());

            Address addr = u.findRelatedOne(Address.class, " or u.id like ?", param("abcd%"));
            System.out.println("  Address:" + addr.getId());

            List<UserRole> userRoles = u.findRelatedList(UserRole.class);
            if (userRoles != null)
                for (UserRole ur : userRoles) {
                    System.out.println("  UserRole:" + ur.getUserId() + "," + ur.getRid());
                }

            Object path = new Object[] { UserRole.class, Role.class, RolePrivilege.class, Privilege.class };
            Set<Privilege> privileges = u.findRelatedSet(path);
            if (privileges != null)
                for (Privilege privilege : privileges)
                    System.out.println("  Privilege:" + privilege.getId());
        }
    }
```

先用普通的SQL查询方法eFindAll方法查询出一个List到内存，然后调用ctx的或ActiveRecord的findRelatedOne、findRelatedList等方法进行手工加载，注意：每次调用这些findRelatedXxx方法时，jSqlBox都会向数据库发送一个SQL，所以要慎重使用这种手工加载方式，这是一个要注意的坑(但这是一个明晃晃的坑，不是陷阱，前文已经说过了，jSqlBox不玩魔术，不会因为调用了实体的Getter/Setter方法在后台隐形发送SQL)。

当要跨级查询时，还必须依次把要经过的类写出来，第一个类是自己，可以省略，例如从User到Privilege，经过了以下类：UserRole.class, Role.class, RolePrivilege.class, Privilege.class。

示例七 先加载到内存，然后在内存里进行关联查询
先上代码再解释：

```
EntityNet net = ctx.iQuery(new EntityNet(), User.class, AUTO_SQL);
ctx.iQuery(net, UserRole.class, AUTO_SQL);
ctx.iQuery(net, Role.class, AUTO_SQL);
ctx.iQuery(net, RolePrivilege.class, AUTO_SQL);
ctx.iQuery(net, Privilege.class, AUTO_SQL);
ctx.iQuery(net, Address.class, AUTO_SQL);
ctx.iQuery(net, Email.class, AUTO_SQL);
User u = net.pickOneEntity(User.class, "u2");
System.out.println("User:" + u.getId());

Address addr = u.findRelatedOne(net, Address.class);
System.out.println("  Address:" + addr.getId());

List<Email> emails = u.findRelatedList(net, Email.class);
if (emails != null)
   for (Email e : emails)
      System.out.println("  Email:" + e.getId());

Set<Role> roles = u.findRelatedSet(net, UserRole.class, Role.class);
if (roles != null)
   for (Role r : roles)
      System.out.println("  Roles:" + r.getId());

Object path = new Object[] { UserRole.class, Role.class, RolePrivilege.class, Privilege.class };
List<Privilege> privileges = u.findRelatedList(net, path);
if (privileges != null)
   for (Privilege privilege : privileges)
      System.out.println("  Privilege:" + privilege.getId());
```

这个例子里，先用7个单独的SQL分别加载7个表的内容汇总成一个EntityNet实例，然后在findRelatedOne/findRelatedList等方法查询时，将这个EntityNet实例作为第一个参数传入，这时候系统就会在EntityNet实例中进行查询，而不是每次发送SQL到数据库查询，从而减轻对数据库的压力，这相当于在内存中维护了一个实体容器作为缓存区。那么这种查询方式，系统怎么知道各个对象之间的关联关系的呢？是根据实体的主键、外键值来对比，如果一个实体的外键值正好是它依赖的外键对象的主键值，则系统认为这两个对象之间有关联。jSqlBox借用了外键配置，如果一个外键对于实体查询来说是必需的，但是实际生成DDL脚本时却不想生成这个外键脚本，可以在外键配置中加一个ddl=false选项，详见jDialets项目。
前面的6个示例，实体的关联是基于SQL的，而示例7是一种在内存中进行的查询，是一种NoSQL式查询。

jSqlBox支持动态配置，如果以上各示例中传入的不是User.class、Role.class之类的类类型，而是一个TableModel实例，则系统将会用TableModel中的配置覆盖掉实体类用注解形式定义的缺省配置，详见“动态配置”一节。

同大多数“半自动ORM”工具一样，jSqlBox只支持关联实体对象的查询，不支持将关联的对象回写到数据库，例如级联更新之类看起来很酷的功能，这种设计是为了降低它的复杂性和维护成本。

最后要说明的一点是，实体关联查询因为在内存中因为需要进行复杂的实体主键对比和实体属性装配，个人建议不要滥用实体关联查询，如果用SQL或ActiveRecord能简单地查询出结果，就没必要通过实体关联的方式来查询。

08-2 jSqlBox的主从表查询

jSqlBox的主从表查询

jSqlBox的主从表查询是5.0.15版起新增的功能，将参数内嵌式SQL查询写成树状，即可实现类似GraphQL的结构化查询，输入和输出的树状结构一致，所见即所得。其优点有：
1.只需要编写针对单表查询的SQL，会自动按主从关联列名生成类似“id in (?, ?...?)”的SQL片段，并将最终查询结果组装成主从表树状结构。
2.采用纯Java和原生SQL，功能强，学习成本低。
3.没有直接输出为JSON，而是输出Map/List对象或Java实体对象，查询结果可以在后端继续修改。
4.可以直接利用Java的IDE格式化和语法检查功能，不需要第三方工具。
5.jSqlBox的内嵌式SQL参数、分页、分库分表、拦截器、事务等依然可以直接使用。
6.没有涉及安全、权限功能，无学习成本。安全、权限这些不属于ORM工具的职能，而应由后端的SpringSecurity/Shiro工具包或独立的Serverless/JsonAPI服务来提供。
7.如果结合我写的MyServerless项目使用，可以实现前端直接在html里书写Java、定制主从表多级查询并返回json, 实现类似GraphQL的功能，将业务逻辑前移到前端。
8.性能好，用"in"的方式进行数据库表的关联查询，不存在1+N问题。
9.源码简洁(实现这个功能仅用了300行源码，见GraphQuery.java)
使用示例：

```
    GraphQuery q1 = //
            $("addresstb as addresses", "where id>", que("a1"), " and id<", que("a5"), pagin(1, 10), //
                    $1("usertb", key("user"), ms("userId", "id"), $("userroletb as userRoleList", ms("id", "userId"), //
                            $("roletb as roleList", ms("rid", "id"), // ms方法也可以写成DB.masterSlave()，它的参数是主、从表的键名
                                    $("roleprivilegetb as rolePrivilegeList", ms("id", "rid"), //
                                            $1("privilegetb as privilege", ms("pid", "id")) //
                                    )//
                            )//
                    ), //
                            $1("select * from emailtb as email", ms("id", "userId")), //
                            $("addresstb as addressList", ms("id", "userId"), "and addressName like ?", par("addr%"))//
                    )//
            );
    GraphQuery q2 = //
            $("usertb as u", "where id>", que("u2"), pagin(1, 10), entity(User.class), //
                    $1("emailtb as emailMap", ms("id", "userId")), //
                    $("addresstb as addressList", ms("id", "userId"))//
            );
    Object result = DB.graphQuery(q1, q2); //result是查询结果
    String json = JsonUtil.toJSONFormatted(result); //输出为JSON文本
```

以上示例详见单元测试下的GraphQueryTest.java，示例结果输出如下：

```
{
  "addresses":[
    {
      "addressName":"address2",
      "id":"a2",
      "userId":"u2",
      "user":{
        "id":"u2",
        "userName":"user2",
        "userRoleList":[
          {
            "id":"3i6yaxy2fusjkgisyfhypkti9",
            "rid":"r1",
            "userId":"u2",
            "roleList":[
              {
                "id":"r1",
                "roleName":"role1",
                "rolePrivilegeList":[
                  {
                    "id":"b484ze4k44xemtkstehnprhxq",
                    "pid":"p1",
                    "rid":"r1",
                    "privilege":{
```

```
                    "id":"p1",
                    "privilegeName":"privilege1"
                  }
                }
              ]
            }
          ]
        },
        {
          "id":"e41dln9m4jehmc7somvu5s2pf",
          "rid":"r2",
          "userId":"u2",
          "roleList":[
            {
              "id":"r2",
              "roleName":"role2",
              "rolePrivilegeList":[
                {
                  "id":"dhrh5kgsod6w76e6xtl36u8b9",
                  "pid":"p1",
                  "rid":"r2",
                  "privilege":{
                    "id":"p1",
                    "privilegeName":"privilege1"
                  }
                },
                {
                  "id":"b9h2aenn6jjacns9ng5vwhaiq",
                  "pid":"p3",
                  "rid":"r2",
                  "privilege":{
                    "id":"p3",
                    "privilegeName":"privilege3"
                  }
                }
              ]
            }
          ]
        },
        {
          "id":"994a5o65pfa7wx8vq99gi1lkg",
          "rid":"r3",
          "userId":"u2",
          "roleList":[
            {
              "id":"r3",
              "roleName":"role3",
              "rolePrivilegeList":[
                {
                  "id":"7qf9us50mw95hijwkfvuzus4q",
                  "pid":"p3",
                  "rid":"r3",
                  "privilege":{
                    "id":"p3",
                    "privilegeName":"privilege3"
                  }
                }
              ]
            }
          ]
        }
      ],
      "email":{
        "emailName":"email3",
        "id":"e3",
        "userId":"u2"
      },
      "addressList":[
        {
          "addressName":"address2",
          "id":"a2",
```

```
            "userId":"u2"
          }
        ]
      }
    },
    {
      "addressName":"address4",
      "id":"a4",
      "userId":"u4",
      "user":{
        "id":"u4",
        "userName":"user4",
        "userRoleList":[
          {
            "id":"bb2d1kuwvii0gpa0pxgaph8zr",
            "rid":"r1",
            "userId":"u4",
            "roleList":[
              {
                "id":"r1",
                "roleName":"role1",
                "rolePrivilegeList":[
                  {
                    "id":"b484ze4k44xemtkstehnprhxq",
                    "pid":"p1",
                    "rid":"r1",
                    "privilege":{
                      "id":"p1",
                      "privilegeName":"privilege1"
                    }
                  }
                ]
              }
            ]
          }
        ],
        "addressList":[
          {
            "addressName":"address4",
            "id":"a4",
            "userId":"u4"
          }
        ]
      }
    }
  ],
  "u":[
    {
      "id":"u3",
      "userName":"user3",
      "addressList":[
        {
          "addressName":"address3",
          "id":"a3",
          "userId":"u3"
        }
      ],
      "emailMap":{
        "emailName":"email5",
        "id":"e5",
        "userId":"u3"
      }
    },
    {
      "id":"u5",
      "userName":"user5",
      "addressList":[
        {
          "addressName":"address5",
          "id":"a5",
          "userId":"u5"
```

```
        }
      ]
    }
  ]
}
```

用法详解：

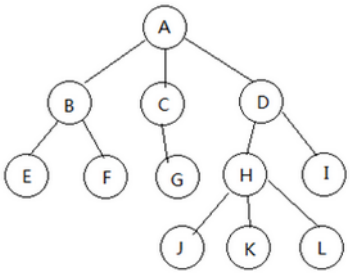
```
* 每个数据库表格对应用一个SQL查询，写在$()或$1()方法中
* $()方法的第一个参数如果没有空格，则系统自动转换为 select * from xxx
* $()方法的第一个参数如果有空格，如"select id, name from tb"，则系统不转换
* $()方法的第一个参数的最后一个单词，将作为输出结果的键名。但是键名也可以用key("键名")来手工指定。
* ms()方法也可以写成DB.masterSlave(),它的参数是主表和从表的键名，参数个数必须是2的倍数，
  ms()支持复合主键，如ms("m1", "m2", "c1","c2" )表示主表的(m1，m2)列关联到从行的(c1,c2)列,
  主表还是从表的判定与数据库定义无关，而是:如果一个$()方法写在另一个$()方法里，则它就是从表,
    ms()方法会被编译成 " where xxId in (?, ?,...,?) " 片段。问号是根据主表的所有关联列值填充为SQL参数
* $1()方法表示仅输出单个元素而不是一个列表，$1()也可以写成$("xxx", "some sql", DB.one)
* 从第二个参数起，即可使用jSqlBox的内嵌sql式语法，普通文本解析为SQL片段，pagin、par、que等方法都可以使用
* 缺省情况下，输出结果为Map/List结构，但是如果出现DB.entity(XxxClass)参数后，这个SQL的输出结果被转换为一
    个实体Bean对象。实体Bean也可以嵌套从表的内容，但是要注意Bean里要有相应的字段定义。
* 使用DB.graphQuery($(), $()...)可以对一个或多个$()方法进行查询。
* 输出对象需要输出为Json时，需要使用者自行在pom中添加JSON工具依赖，并手工进行转换，jSqlBox中并不提供JSON工具
```

09 树结构查询

树结构查询

树结构查询严格来说不属于jSqlBox的特性，只是它的实体关联查询功能的使用示例

假设数据库里有这么一棵树：



id	comments	pid
A	found a bug	
B	is a worm?	A
E	no	B
F	is a bug	B
C	oh, a bug	A
G	need solve it	C
D	careful it bites	A
H	it does not bite	D
J	found the reason	H
K	solved	H
L	uploaded	H
I	well done!	D
END	End Tag	

它的实体类定义:

```
@Table(name = "treenodetb")
public class TreeNode extends ActiveRecord<TreeNode> {
   @Id
   String id;

   @SingleFKey(ddl = false, refs = { "treenodetb", "id" })
   String pid;

   TreeNode parent;

   List<TreeNode> childs;

   //getter & setter...
}
```

@SingleFKey的设置表示pid列参考外键为本身表的id列，ddl=false表示如果利用jDialects创建DDL脚本，将忽略这个外键设定。

在jSqlBox中用以下代码查询，就可以在内存中获得装配好的树结构对象，就这么简单 ：

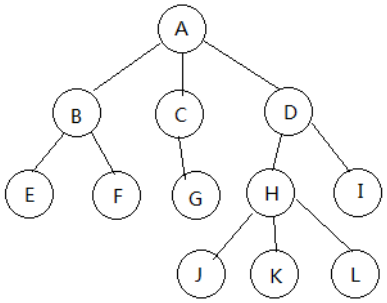
```
   private static final Object[] targets = new Object[] { new EntityNetHandler(), TreeNode.class, TreeNode.class,
         alias("t", "p"), give("p", "t", "parent"), give("t", "p", "childs") };

   @Test
   public void testSearchTreeChild() {
      EntityNet net = ctx.qry(targets, "select t.**, t.pid as p_id from treenodetb t");
      node = net.pickOneEntity("t", "D"); //或pickOneEntity(TreeNode.class,"D");
      ...
   }
```

pickOneEntity方法可以根据主键筛选出任一单个实体，pickOneEntity("t", "D")即表示选取树中的"D"节点。

一个SQL查询出子树

上例中有一个问题，它一次将整棵树加载进了内存。如果一棵树很大，有上百万个节点，则这种做法显然是极慢、极耗内存的，而我们的目标是希望一次只加载一棵树的某个子树，怎么做到? 这就牵扯到一个子树查询的问题。如何高效地查询出一棵子树，有许多做法，有利用树据库本身递归SQL查询功能的、有利用路径枚举(Path Enumerations)方案的，有利用闭包表的(Closure Table)、有利用嵌套集(Nested Sets,又称为左右值法)的。这里我介绍本人发明的一种深度树存储方案(或称为海底捞算法)，可以利用一句SQL无递归查询出任意节点的子树，它的原理请详见基于前序遍历的无递归的树形结构的数据库表设计

首先要对数据库增加两个字段line和lvl，并让数据库按前序遍历顺序存储，并且在整个表的结束添加一个End标记行，具有最大的行号值，原理图见下：



line	id	comments	level
1	A	found a bug	1
2	B	is a worm	2
3	E	no	3
4	F	is a bug	3
5	C	oh, a worm	2
6	G	solve it	3
7	D	careful it bites	2
8	H	Is a bug, not worm	3
9	J	found the reason	4
10	K	solved	4
11	L	uploaded	4
12	I	well done!	3
13	END	end tag	0

对实体类定义改进如下，新增line和lvl两个字段:

```
@Table(name = "treenodetb")
public class TreeNode extends ActiveRecord<TreeNode> { //getter & setter 略
    @Id
    String id;

    @SingleFKey(ddl = false, refs = { "treenodetb", "id" })
    String pid;

    TreeNode parent;

    List<TreeNode> childs;

    Integer line; //行号

    Integer lvl;  //深度

    //getter & setter...
}
```

然后用以下的实体关联查询就可以一句SQL查询并加载任意节点的子树:

```
EntityNet net = ctx.qry(targets,
        "select t.**, t.pid as p_id from treenodetb t where t.line>=? and t.line< (select min(line) from treenodetb where line>? and lvl<=?) ", par(d.g
        TreeNode node = net.pickOneEntity("t", d.getId());
        ...
    }
```

这种深度树方案因为查询时没有用到递归，是所有树结构查询方案中速度最快的方案，但使用它的前题是必须保持树结构在数据库中先排好序，当进行节点移动后必须重新整理一遍，保持排序（删除节点时不需要，因为没有打乱排序)。行号允许跳号，例如按100000跳号，这样插入新值时有可能不必将它后面的所有行号+1，只需要让新行号插在前后行号中间即可。End标记在这种方案是必须的，它具有最大的行号，可以设成一个极大的固定值。
本文示例的源码位于单元测试的EntityNetTreeTest.java。

10 支持重构的SQL

SQL是文本格式，不支持字段的重构，为了利用IDE来检查列名拼写错误，即实现支持重构的SQL，其它持久层工具往往用Java方法代替SQL关键字,造成重复发明SQL这种反模式，可读性差、处理不了复杂逻辑、不利于移植。 jSqlBox不重新发明SQL, 为了让SQL支持重构，下面是几种写法示例，示例源码位于项目单元测试的Java6EampleTest.java和Java8EampleTest.java中:

写法1, 利用ThreadLocal写出支持重构的SQL（适用于旧版jSqlBox，不推荐）

```
User u = AliasProxyUtil.createAliasProxy(User.class, "u"); //"u"可省略
//下面的clean()方法必须，用于清除ThreadLocal内容
List<Map<String, Object>> list = ctx.qryMapList(clean(), //
  "select "//
   , alias(u.getId())//
   , ", ", alias(u.getAddress())//
   , ", ", alias(u.getName())//
   , " from ", table(u), " where "//
   , col(u.getName()), ">=?", param("Foo90") //
   , " and ", col(u.getAge()), ">?", param(1) //
  );
```

写法2, 利用Java8的Lambda语法写出支持重构的SQL（适用于旧版jSqlBox，不推荐）

```
User u = AliasProxyUtil.createAliasProxy(User.class);
List<?> list1 = ctx.qryMapList( //不需要Clean
  "select "//
  , (ALIAS) u::getId//
  , (C_ALIAS) u::getAddress //
  , (C_ALIAS) u::getName //
  , " from ", table(u), " where "//
  , (COL) u::getName, ">=?", param("Foo90") //
  , " and ", (COL) u::getAge, ">?", param(1) //
 );
```

它的运行原理是创建一个代理类，每次访问实体字段时，不是返回值，而是返回字段名本身，并通过种种手段把它对应的数据表列名返回给调用方法去拼接SQL文本。

```
从3.0.0版起，新境了一个名叫"JAVA8"的类，其中有几种静态简化命名方法如$、a$、c$，例如:
(COL) u::getId     可以写成   $(u::getId),
(ALIAS) u::getId   可以写成   a$(u::getId),
(C_ALIAS) u::getId 可以写成   c$(u::getID)，
并有一个pure$方法表示仅返回列名。

上例Java8示例也可写为：
User u = AliasProxyUtil.createAliasProxy(User.class);
List<?> list1 = qryMapList("select ",
        $(u::getId), c$(u::getAddress), c$(u::getName),
        " from ", table(u),
 " where ", $(u::getName), ">=?", param("Foo90"),
        " and ", $(u::getAge), ">?", param(1));
```

写法3 返朴归真，利用字符串常量拼接SQL(推荐)

另一种写出支持重构的SQL的简单做法是将数据表字段定义为字符串常量，在SQL中用这个常量来代替，这种做法不需要复杂的技术,而且从5.0.4版开始，jSqlBox自带了一个可以自动根据数据库生成带常量声明的实体类，详见jDialects文档，利用字符串常量拼接SQL这个就是字面的意思，就不多介绍了。

写法4 利用Q类(推荐)

从5.0.4版开始，jSqlBox自带了一个可以自动根据数据库生成Q类的功能，连表名都可以支持重构了。详见jDialects文档。使用示例如下，对当前数据库所有表自动生成为Q类源码:
TableModelUtils.db2QClassSrcFiles(ctx.getDataSource(), ctx.getDialect(), "c:/temp", "com.demo", "Q");
配置好路径后，每当数据库变动，运行上面语句就会自动生成整个数据库的Q类，下面是生成的一个Q类源码类示例:

```
public class QUserDemo {//注意这个文件是jSqlBox根据数据库自动生成的
 public static final QUserDemo instance = new QUserDemo();
 public String toString(){
  return "user_demo";
 }
 public final String id = "id";
 public final String user_age = "user_age";
 public final String user_name = "user_name";
}
```

则以下不支持重构的SQL:

```
DB.exe("insert into user_demo (id, user_name, user_age)", par(1,"张三"，15),valuesQuestions());
```

可以写成支持SQL重构的形式，在需要表名和字段名的地方，用Q类实例和Q类的常量字段代替即可：

```
QUserDemo u = QUserDemo.instance;
DB.exe("insert into ", u, " (", //
                u.id, ",", par(1), //
                u.user_name, ",", par("张三"), //
                u.user_age, par(15), //
                ")", valuesQuestions());
```

上例位于demo\jsqlbox-qclass目录下。 jSqlBox支持重构的SQL写法相比与JOOQ之类的工具类来说，优点是秒懂、无任何学习负担，因为jSqlBox无非就是手工在Java里拼接SQL字符串和参数而已，没有重新发明SQL。
在示例项目中，还有演示如何定制生成的常量类，比如采用驼峰式字段命名，有兴趣定制的可以自行查看。

写法4 利用JPA的querydsl-maven-plugin插件生成Q类(推荐，但仅适用于JPA实体类混用的场合)

QueryDSL有一个Maven插件，可以扫描项目已有的JPA实体类，生成它的Q类，jSqlBox借花献佛也可以把它的Q类拿过来直接用于拼接原生SQL。
jSqlBox使用这个插件的步骤：
1.在maven里配置好querydsl-maven-plugin插件, 详见demo\jsqlbox-qclass项目的pom.xml配置
2.每当更改过实体类后，运行mvn eclipse:eclipse即可自动生成Q类，注意这个插件生成的Q类位于target目录下。
3.在初始化jSqlBox时，必须手工配置一个SqlItemHandler用来解析Q类，毕竟QueryDSL不是给jSqlBox设计的，要作转换才能用。 4.然后就可以在jSqlBox的SQL里使用QueryDSL的Q类了, 使用示例如下：

```
QUser u = QUser.user;
DB.exe("insert into ", u, " (", //
          u.id, ",", par(1), //
          u.userName, ",", par("张三"), //
          u.userAge, par(15), //
       ")", valuesQuestions());
```

可以看到，在需要表格名的时候，直接传入Q类实例，在需要SQL列名的时候，传入Q类的常量字段即可。

11 多行SQL的存放

多行SQL的存放

jSqlBox推荐将SQL文本直接写在方法参数里，对于简单的SQL、参数多的SQL、复杂的动态拼接SQL场合都适用, 但是当SQL非常长、参数又很少的情况下，可以考虑将SQL单独存放在文件里集中管理，有以下几种做法：

1. 放在Java源码里，作为普通字符串定义，甚至用注解标注在方法体上(比如MyBatis的@Select注解)
2. 放在XML之类的单独的模板文件中。
3. 使用Kotlin之类的支持多行文本块的其它JVM语言。

以上几种做法要注意的是: 1.Java从13版才开始支持多行文本块，而目前有不少项目还在使用Java8版本，无法利用上Java本身的多行文本特性。
2.用模板文件存放SQL，主要问题是：
    1)必须在Eclipse或Idea里安装IDE插件,否则无法实现在IDE里快速定位到SQL。
    2)容易将DAO的功能与模板语法绑定，不利于项目移植。
    3)如果采用XML，通常存在打字繁琐的问题。
3.使用其它JVM语言会增加维护成本。

如果开发环境是Java13版或以上，jSqlBox当然推荐直接将长文本放在Java源码里，这样可以利用Java本身的功能支持长文本变量的重构和快速定位。如果开发环境是Java13以下，可以考虑采用jSqlBox中的Text类以支持多行文本定位及ID重构, 示例如下(位于jSQLBox/core/src/test/resources/text/TextTest.java)：

```
public class TextTest extends TestBase {
    {
        this.regTables(Demo.class);
    }

    public static class InsertDemoSQL extends Text {
        /*-
        insert into demo
        (id, name)
        values(?, ?)
        */
    }

    public static class UpdateDemoSQL extends Text {
        /*-
         update demo
         set name=#{d.name}
         where id=:d.id
        */
    }

    public static class SelectNameByIdSQL extends Text {
        /*-
        select name from demo
            where id=?
        */
    }

    @Test
    public void test() {
  DB.exe(new InsertDemoSQL(), par("1", "Foo"));
  Demo d = new Demo().putField("id", "1", "name", "Bar");
  DB.exe(DB.TEMPLATE, UpdateDemoSQL.class, bind("d", d));
  Assert.assertEquals("Bar", DB.qryString(SelectNameByIdSQL.class, par("1")));
    }

}
```

示例中将Sql多行文本利用Java的/*- */注释来存放，为每个多行文本定义一个类，继承于Text类，它的toString()方法被重载，会自动读取源码中的自身类中的多行文本。jSqlBox的缺省内嵌SQL式写法允许接收Text类的子类实例、或类类型本身作为参数。 Text类不光可以存放SQL,还可以存放其它用途的多行文本。

**注意：利用Java的/*- /注释来实现多行文本支持是个黑科技，详细介绍请见这里，优点是可以利用IDE快速定位到SQL文本。 缺点是需要将包含多行SQL文本的Java文件放在Resources目录而不是直接放在src目录里，并且要在pom.xml的plugins节点下添加如下配置:*

```
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>build-helper-maven-plugin</artifactId>
      <version>1.12</version>
      <executions>
        <execution>
          <id>add-source</id>
          <phase>generate-sources</phase>
          <goals>
            <goal>add-source</goal>
          </goals>
          <configuration>
            <sources>
              <source>src/main/resources</source>
            </sources>
          </configuration>
        </execution>
        <execution>
          <id>add-test-source</id>
          <phase>generate-sources</phase>
          <goals>
            <goal>add-test-source</goal>
          </goals>
          <configuration>
            <sources>
                <source>src/test/resources</source>
          </sources>
          </configuration>
        </execution>
      </executions>
    </plugin>
```

jSqlBox/core项目本身就配置成了支持多行SQL文本，所以可以把它当成一个配置示例来学习。jSqlBox项目单元测试在Maven、Eclipse和Idea开发环境下都实测通过。

12 按实体样板查询

注：按实体样板查询不是太常用，而且局限性很大，初学者可以跳过这一节

按实体样板查询，是指给定一个实体Bean作为样板，然后由系统自动根据这个样板生成SQL，从而从数据库中加载与这个样板类似的对象，示例如下

```
//ctx.setAllowShowSQL(true); 打开SQL日志输出看一下
User u1=new User();
u1.setId(1);
u1.setName("Tom");
u1.setAddress(null);
List<User> users=ctx.entityFindBySample(u1);
```

系统会自动根据样板u1的内容，生成一个类似 “select * from users where id=? and name=?”的SQL并将1、"Tom"两个参数代入，从数据库加载符合条件的记录，返回一个List集合。
从生成的SQL可以看出，它自动生成的条件都是"="号，并且用"and"连接，而且忽略了null值，这是个很大的缺陷，如果遇到复杂的样板条件怎么办?不用担心，复杂的条件平时是不可能遇到的，用entityFindBySample可以解决所有问题。

开个玩笑，复杂的样板条件jSqlBox也是有点办法的，见下面几个示例：

```
User u= new User("Nam", "addr");
List<User> users = ctx.entityFindAll(User.class, new SampleItem(u).sql(" where  ").notNullFields());

List<User> users = ctx.entityFindAll(User.class, new SampleItem(u).sql(" where (").allFields()
            .sql(") or name like ?").param(":name%").sql(" order by name"));

List<User> users = ctx.entityFindBySQL(User.class, new SampleItem(u).sql("select * from users where (")
   .nullFields().sql(") or name like ?").param(":name%").sql(" order by name"));

List<User> users = u.findAll(new SampleItem(u).sql(" where  ").notNullFields());
```

SampleItem是一个继承于CustomizedSqlItem的类，它是一个自定义Sql条目，专门用于将样板转化为SQL片段。它的SQL方法直接添加一段SQL文本，它的param方法会添加一个SQL参数，并将其中的:xxxx占位符替换成样板中的属性值。 allFields方法将会把样板中的所有属性自动拼成一段" fieldxx=? and fieldsxx=? and fieldxxx is null" 之类的SQL及参数片段。 notNullFields方法则类似地拼一段SQL,但是忽略所有样板中为null的属性。 nullFields方法则是只拼接属性为null的属性成一段“ fieldxxx is null and fieldxxxx is null...”这样的SQL片段。

样板通常用于根据实体样板创建简单的SQL，实际上SampleItem的源码也很简单，只有130行而已（这从侧面也展示了jSqlBox的可扩充性好）。对于非常复杂的逻辑，建议还是用回普通的SQL查询语句来解决问题。因为当逻辑非常复杂时，依靠样板来拼接SQL，可读性很差。

13 Handler类介绍

jSqlBox中的Handler类分为两大类： ResultSetHandler结果集处理类和SqlHandler拦截器类，虽然都叫Handler类，用法也有点相像，但它们的作用是不一样的，不能混为一谈。
Handler类直接使用的频率不高，但它是jSqlBox的底层技术基础，复杂的功能往往都是利用Handler拦截器类来对SQL或参数进行变换来实现的。

ResultSetHandler 结果集处理类

ResultSetHandler是一个从DbUtils开始就支持的接口类，在jSqlBox的许多SQL方法中(主要是从QueryRunner中继承下来的)都可以将ResultSetHandler类实例作为参数传入，例如ctx.qryMapList方法实际就类似于下面的调用:

```
List<Map<String, Object>> result =
  ctx.qry("select u.* from DemoUser u where u.age>?", new MapListHandler(),0);
```

在DbUtils中支持的ResultSetHandler类实现：

```
ArrayHandler  把结果集中的第一行数据转成对象数组
ArrayListHandler  把结果集中的每一行数据都转成一个数组，再存放到List中
BeanHandler  将结果集中的第一行数据封装到一个Bean实例中
BeanListHandler 将结果集中的每一行数据都封装到一个Bean实例中，存放到List里
BeanMapHandler  将结果集中的每一行数据都封装到一个Bean实例中，按指定列存放到Map里
ColumnListHandler 将结果集中某一列的数据存放到List中
KeyedHandler    将结果按指定主键装配成一个Map<Object, Object>类
MapHandler    将结果集中的第一行数据封装到一个Map里
MapListHandler  将结果集中的每一行数据都封装到一个Map里，然后再存放到List
ScalarHandler  返回一个单独的值类型，即查询结果第一行第一列的值
```

具体每个ResultSetHandler类的详细介绍请参见DbUtils的使用手册。 另外在jDbPro模块中增加了一个TitleArrayListHandler, 将结果返回一个List<Object[]>类型，它与ArrayListHandler的区别是第一个List元素是一个数组，内容是各个列的标题。

SqlHandler拦截器类

(拉截器类按英文命名应该为Interceptor，但是因为Interceptor这个命名已经被很多项目用滥了，为了避免不必要的重名，所以在jSqlBox中将其命名为SqlHandler)

SqlHandler拦截器类是一个接口，一个方法里可以传入多个SqlHandler拦截器，形成一个拦截器链。要注意的是，SqlHandler拦截器对于从QueryRuner系列直接继承的SQL方法不起作用, 也就是那些需要手工捕获SqlException的方法，因为这些方法已经非常底层了，它们是jSqlBox的底层函数，很少会直接使用，即没必要，也很难再添加拦载器了。

jSqlBox中的SqlHandler拦截器类实现主要有：

```
PrintSqlHandler：这是一个调试用拦截器，打印输出SQL到控制台
SimpleCacheHandler：这是一个缓存工具，利用内存HashMap实现了简单的LRU查询缓存
PaginHandler：这是一个分页拦截器，用于将普通SQL转化为分页SQL，其内部调用了jDialects的分页函数
EntityListHandler: 用于将SQL查询出来的List<Map<String,Object>>结果集装配成一个实体List
SSHandler: 用于将"select u.** ..." 这种非标SQL(简称双星SQL，见下）转为正常SQL
SSMapListHandler: 用于处理双星SQL，并返回List<Map<String,Object>>
SSTitleArrayListHandler: 用于处理双星SQL，并返回List<Object[]>类型，第一个List对象为各个列的标题数组
EntityNetHandler: 用于处理双星SQL, 并将查询结果装配成一个EntityNet，详见实体关联查询一节
CamelHandler: 这是从5.0.6版开始增加的一个拦截器，用来将返回Map结果中的下划线字段名转为驼峰式字段名，如user_name转为userName
```

这里解释一下“双星SQL”，示例如下：

```
EntityNet net = ctx.qry(new EntityNetHandler(), User.class, UserRole.class, "select u.**, ur.** from usertb u left join userroletb ur on u.id=ur.useri
```

"u.**" 这种写法在运行时被翻译成u.name as u_name, u.age as u_age...，这是因为在jSqlBox实体关联查询时，当涉及两个以上实体，为了准确区分不同的实体属性的数据库对应字段，必须在查询时给字段起个别名来区分，用手工打字非常繁琐(也是可以的)，采用双星写法可以极大地简化打字量。类似地u.##在jSqlBox中也会被特别处理，代表只加载所有主键和外键字段而忽略其它字段。

禁用拦截器

在SQL方法最后一个参数传入一个new SqlItem(SqlOption.DISABLE_HANDLERS, xxxHandler.class)作为参数就可以禁用这个类对应的所有拦截器。它的一个应用示例如下：

```
    public void pageQuery(Object... conditions) {
        qryLongValue("select count(1) from sys_user where 1=1 ", conditions, noPagin());
        qryForEntityList(DemoUser.class, "select * from sys_user where 1=1", conditions);
    }
```

上例中noPagin是引入DB类中的静态方法，相当于new SqlItem(SqlOption.DISABLE_HANDLERS, PaginHandler.class),它可以禁用当前拦截器中的分页类。这个例子中用户传来一组参数，其中可能有一个分页拦截器，也可能没有，这个示例中的第一行不管三七二十一，先禁用掉分页拦截器, 才好统计总数，否则SQL会报错。

SqlHandler拦截器还有两种特殊的用法(不推荐):

用法1： 配置成全局拦截器 在DbContext构造时，可以设定一组拦截器，这组拦截器将会在所有SQL执行时被调用:

```
DbContext ctx=new DbContext(ds);
CacheHandler cache=new SimpleCacheHandler();
ctx.setSqlHandlers(new PrintSqlHandler(), cache);
```

则所有DbContext方法执行时都将打印SQL输出，并开启了查询缓存。
全局拦截器影响的范围比较大，所以如果没有特殊原因，一般不建议配置全局拦截器。

用法2：线程局部变量拦截器 利用线程局部变量，可以随时给任意SQl方法强行添加一组SqlHandler拦截器，而无需作为方法体的参数传入，如下例用法可以给一个普通查询强行加入分页和打印SQL功能：

```
ctx.setThreadLocalSqlHandlers(new PaginHandler(1, 5), new PrintSqlHandler() );
try{
 ctx.qryMapList("select * from users");
} finally{
 ctx.getThreadedHandlers().clear();
}
```

注意：虽然jSqlBox在每次SQL执行后会自动清空线程局部变量拦截器，但为安全起见(如异常发生情况下)，这种用法必须用try...finally来确保清除线程局部变量，以避免线程局部变量泄漏造成后面的程序出现严重的逻辑错误。

自定义SqlHandler拦截器

自定义拦截器类在jSqlBox中是很简单的，只需要实现SqlHandler接口或继承于DefaultOrderSqlHandler类即可，前者必须实现getOrder方法，后者用于对执行顺序没有要求的场合。当有多个拦截器同时存在时，如果order值设的越小，则执行顺序越靠前，DefaultOrderSqlHandler类的order值缺省是100。

例如自定义一个拦截器类，在SQL执行前打印一段"Hello",结束后打印"Bye"，示例如下：

```
public static class MyDemoAroundSqlHandler extends DefaultOrderSqlHandler {
  @Override
  public Object handle(ImprovedQueryRunner runner, PreparedSQL ps) {
     System.out.println("Hello");
     Object result = runner.runPreparedSQL(ps);
     System.out.println("Bye");
     return result;
   }
 }
```

上例中的参数ImprovedQueryRunner是在jDbPro项目中的定义，是DbContext的父类，在jSqlBox环境下，参数ImprovedQueryRunner可以强制转换为DbContext类型使用。

最后即可在程序中使用自定义拦截器:

```
List<Map<String, Object>> result2 = qry(new MapListHandler(), new MyDemoAroundSqlHandler(), "select u.* from DemoUser u where u.age>?",
```

14 jSqlBox配置详解

jSqlBox所有功能都基于DbContext展开，DbContext是一个无会话的(Sessionless)轻量级对象，最简单的使用方式是用new DbContext()或new DbContext(DataSource)方法创建一个实例，这样对于新手来说可以很快上手使用，但是对于需要改变默认配置、提供更多功能的情况下(如事务管理、日志输出等)，就必须进行更多配置。DbContext的配置有两种方式:

1. 直接用set方法进行每个DbContext的设定

使用示例如下：

```
DbContextctx=new DbContext(ds);
ctx.setDialect(Dialect.H2);//设定方言
ctx.setAllowShowSQL(true); //允许日志输出
```

DbContext是个伪线程安全对象，它的所有以set打头的方法不是线程安全的，但是只要保证这些set方法只在程序启动时调用一次，就不会影响到DbContext实例的线程安全性。
所有配置方法如下：

```
setAllowShowSQL(Boolean) //启或关闭日志输出功能，见下
setBatchSize(Integer) //设定批处理默认缓存条数，见下
setConnectionManager(ConnectionManager)//设定连接管理器，见“事务配置”
setMasters(DbPro[]) //设定一组主库，见“分库分表”
setMasterSlaveOption(SqlOption) //设定主从库访问策略，见“分库分表”
setName(String) //给自己设定一个名字，见“分库分表”
setSlaves(DbPro[]) //设定一组从库，见“分库分表”
setSqlHandlers(SqlHandler[]) //设定一组全局拦截器，见“Handler类介绍”
setSqlTemplateEngine(SqlTemplateEngine) //设定模板引擎，见“tXxxx模板方法”
setDialect(Dialect) //手工指定DbContext的数据库方言，而不是由jSqlBox自动根据DataSource来猜测方言类型
setShardingTools(ShardingTool[]) //设定一个Sharding工具，见“分库分表”
setSnowflakeCreator(SnowflakeCreator) //设定一个分布式主键生成器，见“分库分表”
setAuditorGetter(Object) //注见实体注解的@CreateBy和@LastModifiedBy一节
setJavaToJdbcConverter(JavaToJdbcConverter javaToJdbcConverter);//设定Java转JDBC参数的转换器，缺省是BasicjavaToJdbcConverter.instance
setJdbcToJavaConverter(JdbcToJavaConverter jdbcToJavaConverter);//设定Jdbc结查转Java值的转换器，缺省是BasicJdbcToJavaConverter.instance
setIgnoreNull(boolean) //默认为false，如果设为true,所有实体存入或更新时将忽略值为null的字段，慎用！
setIgnoreEmpty(boolean) //默认为false，如果设为true,所有实体存入或更新时将忽略值为null或空字符串的字段，慎用！
```

setIgnoreNull和setIgnoreEmpty方法要慎用，因为它会在实体存入或更新时忽略什为null或空的属性。一般不打开这两个开关，而是手工在实体插入或更新时加入一个IGNORE_NULL或IGNORE_EMPTY参数，如：
user.insert(DB.IGNORE_EMPTY); 这样虽然麻烦一点，但是能更好地控制实体字段的Null值和空值处理。

2. 调用setGlobalNextXxxx系列静态方法:

调用以下DbContext中的静态方法，也可以达到快速配置目的:

```
setGlobalNextAllowShowSql(Boolean)
setGlobalNextBatchSize(Integer)
setGlobalNextConnectionManager(ConnectionManager)
setGlobalNextIocTool(IocTool)
setGlobalNextMasterSlaveOption(SqlOption)
setGlobalNextSqlHandlers(SqlHandler...)
setGlobalNextTemplateEngine(SqlTemplateEngine)
setGlobalNextDialect(Dialect)
setGlobalNextShardingTools(ShardingTool[])
setGlobalNextSnowflakeCreator(SnowflakeCreator)
setGlobalNextSqlMapperGuesser(SqlMapperGuesser)
setGlobalNextAuditorGetter(Object)
setGlobalNextJavaToJdbcConverter(JavaToJdbcConverter javaToJdbcConverter);
setGlobalNextJdbcToJavaConverter(JdbcToJavaConverter jdbcToJavaConverter);
setGlobalNextIgnoreEmpty(boolean)
setGlobalNextIgnoreNull(boolean)
```

这些方法告诉DbContext默认的全局配置参数是什么。所有方法都是以“setGlobalNext”开头，表示它是一个全局开关方法，会影响到整个项目的下一个DbContext的默认配置，所以使用时要慎重，通常只在程序开始时运行一次，设定好各个默认参数。注意它只影响到下一次DbContext实例的创建，例如在DbContext已经创建好的情况下调用DbContext.setGlobalNextAllowShowSql(true)是不会影响到之前创建的DbContext实例的。

在jSqlBox中，除了日志系统外，所有配置都是通过Java代码完成，对于必须通过读取配置文件更改配置的场合，请自行实现配置文件到Java配置的转换。

在DbContext中还有一个特殊的静态配置

DbContext.setGlobalDbContext(DbContext);
表示设定整个项目的默认全局DbContext实例，通常用于单数据源场合下ActiveRecord实例来使用，例如:
new User().insert();
这种用法，它会调用这个默认全局DbContext实例来进行数据库存取操作。在程序中可以用静态方法DbContext.getGlobalDbContext()或DB.gctx()方法来获取这个默认全局DbContext实例。

DbContext各个设置方法详解

setAllowShowSQL(Boolean) 开启或关闭日志输出

这个设置用来设定是否允许jSqlBox输出日志，默认是关闭。
打开这个开关应该能看到日志输出到控制台，但不一定，还与jSqlBox配置了什么日志工具、以及日志工具本身的配置文件的设置有关，例如，在项目的根目录(对应开发环境的main/resources目录)添加一个文本文件 “jlogs.properties” ，内容如下：
log=com.github.drinkjava2.jlogs.SimpleSLF4JLog
则整个项目将采用SLF4J进行日志输出。

setBatchSize(Integer) 设定批处理默认缓存条数

jSqlBox中有一个批处理开关功能，SQL批量插入、更新语句在内存中缓存起来，然后调用数据库的批处理功能（如果支持的话）一次刷新到数据库，以加快速度，setBatchSize方法用来设定缓存大小，默认值为300。(详见“批处理”一节)

setConnectionManager(ConnectionManager) 设定连接管理器

这个与事务有关，请详见“事务配置”一节

setSqlHandlers(SqlHandler[]) 设定一组全局拦截器

用于给当前DbContext设定一个或多个拦截器，所有基于此DbContext实例上的SQL操作都被拦截器处理，详见“Handler类介绍”。

setSqlTemplateEngine(SqlTemplateEngine) 设定模板引擎

jSqlBox自带一个简易的模板引擎，但是它可以利用setSqlTemplateEngine方法切换模板引擎配置，这个已经在“t系列模板方法”一节介绍过。注意模板引擎不光可以在DbContext中设定，还可以作为pintea系中的方法参数传进去临时使用一下。

在jSqlBox的demo/jsqlbox-beetl目录下有一个示例，演示了在jSqlBox中如何定制自己的SQL模板，这个示例使用Beetl作为模板引擎

setDialect(Dialect) 指定数据库方言

指定DbContext实例的数据库方言，而不是根据DataSource来猜测方言类型。如果在构建DbContext实例时，不指定一个方言，则它会从DataSource中获取一个连接，读取数据库的MetaData, 自动猜测出当前数据库方言类型。(详见jDialects项目)。

setName(String) 给自己设定一个名字

设定当前DbContext实例的名字，某些场合下(如分库分表环境下)，可以利用这个名字来区分不同的上下文实例。

以下为与分库分表相关的配置，详见“分库分表”一节

setMasterSlaveOption(SqlOption) 设定主从库访问策略
setMasters(DbPro[]) 设定一组主库
setSlaves(DbPro[]) 设定一组从库
setShardingTools(ShardingTool[]) 设定一个Sharding工具
setSnowflakeCreator(SnowflakeCreator)设定一个分布式主键生成器实例

15 实体注解配置

实体注解配置只针对个别实体类本身，在jDialects中，已经介绍过。jDialect会根据实体注解生成一个与数据库无关的Java模型,并可以动态生成或修改这个模型，但是如何利用这个模型进行实体到数据库的映射操作是ORM工具的工作。

jDialects本身独有的注解有：
@AutoId,@COLUMN,@CreatedBy,@CreateTimestamp,@FKey,@FKey1,@FKey2,@FKey3,@IdentityId,@LastModifiedBy,@PKey,@ShardDatabase,@ShardTable,@SingleFKey,@SingleIndex,@SingleUnique,@Snowflake,@TimeStampId,@UpdateTimestamp,@UUID25,@UUID26,@UUID32,@UUID36,@UUIDAny,

jDialects自带的与JPA重名的注解有(JPA注解有90多个，jSqlBox只支持以下这15个)：
@Column,@Convert,@Entity,@Enumerated,@GeneratedValue,@GenerationType,@Id,@Index,@SequenceGenerator,@Table,@TableGenerator,@Temporal,@Transient,@UniqueConstraint,@Version

一般的原则是，如果jDialects的注解与JPA完全重名，则表示它的用法与JPA注解完全相同，所以可以参考JPA教材学习即可。而且如果引入了JPA库javax.persistence-api之后，jSqlBox也可以直接识别并使用JPA的上述15个注解。

本节不介绍所有的实体注解，因为大多数实体注解在jDialects的文档中有介绍。这里主要介绍一些略微复杂一点的注解：

@Version 乐观锁注解

它的原理是在数据库表中加一个额外的整数列表示数据版本号，更新时检查版本号是否一致,如果相等,则更新成功,且版本号+1.如果不等,则数据已经被修改过,更新失败。
jSqlBox目前版本只支持使用整数(Integer, Short, Long)类型作为乐观锁字段，不支持使用时间戳。乐观锁使用示例：

```
public static class VersionDemo extends ActiveRecord<VersionDemo> {
    @Id
    @UUID25
    private String id;

    private String name;

    @Version
    private Integer optlock;
    ......

@Test
public void testVersion() {
    VersionDemo v = new VersionDemo();
    v.setName("Foo");
    v.insert();
    v.setName("Bar");
    v.update();
    v.delete();
}
```

以上insert、update、delete方法里，如果发现v已经被修改过，即版本号不同时，会抛出一个运行期异常。通常应用时，运行期异常会配置成被事务切面工具捕获，导致事务回滚。

@Enumerated 枚举字段注解

表示一个实体的属性为枚举类型，可以有EnumType.STRING或EnumType.ORDINAL两种类型，后者为默认类型，可以省略。示例如下：

```
public static class EnumDemoBean extends ActiveRecord<EnumDemoBean> {
    @PKey
    @UUID25
    private String id;

    private String name;

    @Enumerated // EnumType.ORDINAL可省略
    private EnumDemo enum1 = EnumDemo.PART_TIME;

    @Enumerated(EnumType.STRING)
    private EnumDemo enum2 = EnumDemo.PART_TIME;
    ......
```

```
    @Test
    public void testEnum() {
        EnumDemoBean v = new EnumDemoBean();
        v.setName("Foo");
        v.setEnum1(EnumDemo.FULL_TIME);
        v.setEnum2(EnumDemo.FULL_TIME);
        v.insert();
        v.setName("Bar");
        v.setEnum1(EnumDemo.CONTRACT);
        v.setEnum2(EnumDemo.CONTRACT);
        v.update();
    }
```

@Convert 自定义字段转换器

jSqlBox只支持基本类型如String、Integer、Boolean等类型作为实体的属性，如果希望支持复杂对象到数据库列的映射，就必须利用@Convert注解来进行自定义列的转换。使用示例如下：

```
public static class FooDemo {
        public int id;

        public FooDemo(int id) {
            this.id = id;
        }
    }

    public static class FooConverter extends BaseFieldConverter {
        @Override
        public Object entityFieldToDbValue(ColumnModel col, Object entity) {
            Object value = DbContextUtils.doReadFromFieldOrTail(col, entity);
            return ((FooDemo) value).id;
        }

        @Override
        public void writeDbValueToEntityField(Object entityBean, ColumnModel col, Object value) {
            DbContextUtils.doWriteToFieldOrTail(col, entityBean, new FooDemo((Integer) value));
        }
    }

    public static class ConverterDemo extends ActiveRecord<ConverterDemo> {
        @PKey
        @UUID25
        private String id;

        @Convert(FooConverter.class)
        @Column(columnDefinition = "integer")
        private FooDemo foo;
        ......
```

使用：

```
    @Test
    public void testConvert() {
        ConverterDemo v = new ConverterDemo();
        v.setFoo(new FooDemo(1));
        v.insert();
        v.setFoo(new FooDemo(2));
        v.update();
    }
```

@CreateTimestamp 创建时间戳注解

在实体的Date或Timestamp字段上加上这个注解，将会在实体创建时写入当前的时间。

```
    @Temporal(TemporalType.TIMESTAMP)
    @CreateTimestamp
```

```
    java.util.Date  date;
```

这个@CreateTimestamp注解也可以用@COLUMN(createTimestamp = true)来代替。@COLUMN这个全大写的注解是jDialects独有的，它与JPA的@Column注解的区别是其中多了creationTimestamp, updateTimestamp, createdBy, lastModifiedBy, comment, tail这几个字段。

@UpdateTimestamp 修改时间戳注解

在实体的Date或Timestamp字段上加上这个注解，将会在实体创建和修改时写入当前的时间。这个注解也可以用@COLUMN(updateTimestamp=true)代替。

@CreatedBy 创建人注解

在实体的String类型字段上加上这个注解，将会在实体创建时写入当前的编辑者。这个注解也可以用@COLUMN(createdBy=true)代替。

@LastModifiedBy 修改者注解

在实体的String类型字段上加上这个注解，将会在实体创建和修改时写入当前的编辑者。这个注解也可以用@COLUMN(lastModifiedBy=true)代替。

注意在使用@CreatedBy和@LastModifiedBy时，在jSqlBox里还必须在程序启动阶段绑定一个当前编辑者获取类的实例，这个类中必须有一个名为getCurrentAuditor的方法，示例如下：

```
public static class XxxxUserAuditor{
   public Object getCurrentAuditor() {
     //从Spring Security或Shiro之类的单点登录工具中取当前用户ID
     return XxxxSecurityTool.getUserXxxxId();
  }
}

DbContext.setGlobalNextAutidorGetter(new XxxxUserAuditor());
//上行是全局设定，也可以个别设置，如下：
DbContext ctx=new DbContext(ds);
ctx.setAuditorGetter(new XxxxUserAuditor());
```

16 事务配置

jSqlBox采用jTransactions模块作为它的事务工具，jTransactions是一个独立的、与jSqlBox无关的事务工具，在Maven上有单独的发布版。除此之外，jSqlBox还自带一个支持分库分表的分布式事务模块，分布式事务详见"分布式事务"一节，在本节只介绍常规的几种事务管理方式。

第一种方式：手工控制Connection的事务开启和关闭

示例如下：

```
DbContext ctx=new DbContext();
Connection conn= null;
try {
conn= dataSource.getConnection();
conn.setAutoCommit(false);
ctx.update(conn, 'update users set age=?', 5);
      ......
conn.commit();
} catch (Exception e) {
conn.rollback();
} finally {
ctx.close(conn);
}
```

以上方式局限性比较大，因为必须手工捕获Exception，而且方法中必须传入一个Connection参数,使用起来比较烦琐，实际项目中很少采用。

第二种方式: 自动提交模式

这是jSqlBox缺省的事务模式，只要没有进行任何事务有关的配置，所有插入、更新等操作都运行在自动提交模式下。例如：

```
DbContext ctx=new DbContext(someDataSource);
ctx.exe('update users set age=?', 5);
ctx.exe('delete from users');
```

以上两个SQL的执行是独立的，不是在同一个事务下。

第三种方式: 手工控制DbContext实例的事务开启和关闭

```
DbContextctx = new DbContext(dataSource);
ctx.startTrans();
try {
   new User().putField("firstName", "Foo").insert(ctx);
   Assert.assertEquals(101, ctx.entityCount(Tail.class, tail("users")));
   ctx.commitTrans();
} catch (Exception e) {
   ctx.rollbackTrans();
   throw new DbException(e);
}
```

从4.0.7版起，添加了事务模板回调方法，上面的用法也可以写成：

```
DbContext ctx = new DbContext(dataSource);
ctx.tx(()->{
   new User().putField("firstName", "Foo").insert(ctx);
   Assert.assertEquals(101, ctx.entityCount(Tail.class, tail("users")));
 });

//上面的写法是Lambda语法，等效于：
 ctx.startTrans();
 try {
   new User().putField("firstName", "Foo").insert(ctx);
   Assert.assertEquals(101, ctx.entityCount(Tail.class, tail("users")));
   lastTxResult.set(commitTrans());
 } catch (Exception e) {
   lastTxResult.set(rollbackTrans().addCommitEx(e));
```

```
    throw new DbException(e);
  }
```

与tx方法类似的还有一个tryTx方法，如果方法体内有异常发生，它们事务都会回滚。两者的区别是tx方法没有返回值，如果方法体内有异常发生，事务会回滚，并抛出一个运行时异常DbException。而tryTx方法不抛出异常，它返回一个布尔值，返回true表示事务提交成功，返回false表示事务提交失败。 以上两个方法都可以用ctx.getLastTxResult来返回一个TxResult对象，它的commitEx属性里存放了具体发生的异常错误类型，它的result属性里存放了事务提交结果，注意这是一个三态值，有SUCESS、FAIL、UNKNOW三种状态，UNKNOW态很少会遇到，它只发生在使用了jSqlBox的GTX分布式事务，有部分提交且自动回滚也未完成(如网线断掉)这种情形。 tx和tryTx这种事务模板式写法可以用于ManualTxConnectionManager、TinyTxConnectionManager、GroupTxConnectionManager和Gtx这四种事务模块:

1.ManualTxConnectionManager模块
ManualTxConnectionManager不是线程安全的，只能在一个线程中使用，所以必须在每个线程里创建新的DbContext实例并创建一个新的ManualTxConnectionManager实例。

```
DbContextctx = new DbContext(dataSource);
ctx.setConnectionManager(new ManualTxConnectionManager());
ctx.tx(()->{
  new User().putField("firstName", "Foo").insert(ctx);
  Assert.assertEquals(101, ctx.entityCount(Tail.class, tail("users")));
 });
```

这种事务管理方式的优点是可以独立控制每个DbContext的事务，但缺点是必须创建多个DbContext实例。上例完整源码可以参见单元测试下的ManualTxTest.java。

2.TinyTxConnectionManager模块
使用了TinyTxConnectionManager事务连接管理类后，所有的DbContext可以共享同一个TinyTxConnectionManager单例，不再需要单独为每个DbContext实例创建一份事务管理类，使用示例如下：

```
DbContextctx = new DbContext(dataSource);
//ctx.setConnectionManager(TinyTxConnectionManager.instance());//可以省略
ctx.tx(()->{
  new User().putField("firstName", "Foo").insert(ctx);
  Assert.assertEquals(101, ctx.entityCount(Tail.class, tail("users")));
 });
```

TinyTxConnectionManager的限制是在同一个线程里，一个事务只允许一个DbContext获取Connection连接，进行数据库操作，它在底层使用了ThreadLocal类来实现这个功能。上例完整源码可以参见单元测试下的TinyTxTest.java，以及TinyTxTester.java，后者是声明式事务的演示。TinyTxConnectionManager是DbContext的缺省事务管理器，所以上例中的setConnectionManager方法可以省略。

3.GroupTxConnectionManager模块
使用了GroupTxConnectionManager事务连接管理类后，所有的DbContext可以共享同一个GroupTxConnectionManager单例，不再需要单独为每个DbContext实例创建一份事务管理类。它允许一个线程里可以有多个DbContext获取Connection连接，同时进行数据库操作，但是请注意它不能保证每个Connection提交的一致性，也就是说，它不是分布式事务。这个事务模块只能用于一些不重要的场合，如果需要严格的分布式事务，请参见下一节“分布式事务”。GroupTxConnectionManager的使用示例如下

```
DbContextctx1 = new DbContext(dataSource1);
DbContextctx2 = new DbContext(dataSource2);
ctx1.setConnectionManager(GroupTxConnectionManager.instance());
ctx2.setConnectionManager(GroupTxConnectionManager.instance());
ctx1.tx(()->{
  new User().putField("firstName", "Foo").insert(ctx1);
  new User().putField("firstName", "Foo").insert(ctx2);
  Assert.assertEquals(101, ctx1.entityCount(Tail.class, tail("users")));
});
```

上例完整源码可以参见单元测试下的GroupTxTest.java，以及AnnotationGroupTxTest.java，后者是声明式事务的演示。

4.GtxConnectionManager分布式事务模块 因为分布式事务比较复杂，它的介绍请详见分布式事务一节。

另外还有SpringConnectionManager和JFinalConnectionManager两个模块，分别用于Spring和JFinal场合，就不作详细介绍了。它们在底层调用了Spring或JFinal的事务功能,相当于一个适配器。

第四种方式：声明式事务

以上是基本的手工开启和提交事务的介绍，实际项目中还可以用TinyTxAOP、GroupTxAOP、GtxAOP等进行事明式事务的配置，例如以下为一个声明式

事务的配置完整演示，注意这个示例使用了jSqlBox自带的IOC/AOP工具，不需要Spring的支持：

```
public class AnnotationTxDemoTest {
    public static class DataSourceCfg extends BeanBox {
        {
            setProperty("jdbcUrl", "jdbc:h2:mem:DBName;MODE=MYSQL;DB_CLOSE_DELAY=-1;TRACE_LEVEL_SYSTEM_OUT=0");
            setProperty("driverClassName", "org.h2.Driver");
            setProperty("username", "sa");
            setProperty("password", "");
        }

        public HikariDataSource create() {
            HikariDataSource ds = new HikariDataSource();
            this.setPreDestroy("close");// jBeanBox will close pool
            return ds;
        }
    }

    @Retention(RetentionPolicy.RUNTIME)
    @Target({ ElementType.METHOD })
    @AOP
    public static @interface TX { // This is a customized AOP annotation
        public Class<?> value() default TinyTxAOP.class;
    }

    DbContext ctx;
    {
        ctx = new DbContext((DataSource) BeanBox.getBean(DataSourceBox.class));
        ctx.setConnectionManager(TinyTxConnectionManager.instance());
    }

    @TX
    public void txInsert() {
        ctx.exe("insert into user_tb (id) values('123')");
        System.out.println(1 / 0); // DIV 0!
    }

    @Test
    public void doTest() {
      AnnotationTxDemoTest tester=JBEANBOX.getBean(AnnotationTxDemoTest.class);
      tester.txInsert();//事务出错，自动回滚
      ...
    }

}
```

第五种方式：使用Spring的声明式事务

上例中使用了jBeanBox这个微型IOC/AOP工具，这在jSqlBox4.0.0版起已内含。jBeanBox采用了Aop alliance联盟标准接口，所以上例也可以很容易切换成使用常见的Guice或Spring作为IOC/AOP工具。如果使用集成化的Spring-boot环境，配置可以简化成如下：

```
@SpringBootApplication
public class JsqlboxInSpringbootApplication {
    @Autowired
    DataSource ds;

    public static void main(String[] args) {
        SpringApplication.run(JsqlboxInSpringbootApplication.class, args);
    }

    @Bean
    public DbContext createDefaultDbContext() {
        DbContext ctx = new DbContext(ds);
        ctx.setConnectionManager(SpringTxConnectionManager.instance());
        DbContext.setGlobalDbBoxContext(ctx);// 设定静态全局上下文
        return ctx;
    }
}
```

上例请详见demo目录下的jsqlbox-springboot演示项目。注意这个示例的配置不光使jSqlBox可以单独使用，还支持jSqlBox和Hibernate或MyBatis混用在Spring环境中。

17 主从分库分表多租户

主从分离

jSqlBox支持主从分离，DbContext与此相关的配置方法有：

```
setSlaves(DbContext[]) 设定一组从库
setMasterSlaveOption(SqlOption) 设定主从库访问策略
```

对于每个DbContext实例，都可以设定一组从库，并用setMasterSlaveOption方法来设定它们的访问策略，其参数有以下几个选项：

```
SqlOption.USE_MASTER 读写都在当前主库
SqlOption.USE_AUTO   写操作只在当前主库上，读操作在从库上随机选一个，但是当事务开启时(jTransactions中的TinyTx事务或Spring事务)，读操作也在主库上
SqlOption.USE_BOTH   写操作在当前主库和所有从库(注意是所有从库！)，读操作只在主库上
SqlOption.USE_SLAVE  写操作在所有从库(注意是所有从库！)，读操作则在从库上随机选一个
```

如果配置了从库(setslaves)，jSqlBox的默认选项是USER_AUTO策略

注意以上所说的主库、从库是相对于当前DbContext实例来说的，因为每个DbContext只能配置一个数据源，正好对应一个数据库，这里的库，即相当于DbContext实例。

另外，SqlOption的几个选项，也可以直接作为参数传递给jSqlBox的SQL方法，强制控制每个SQL方法到底运行在哪个策略下，如下为一些示例（源码见单元测试的MasterSlaveTest.java), 通常选项用静态引入方式使用:

```
//Auto模式,写在主库上
ctx.exe("update TheUser set name=? where id=3", "NewValue");

//Auto模式，随机读一个从库
TheUser u1 =new TheUser().useContext(ctx).put("id", 3L).load();

//强制模式, 写在所有从库上
ctx.exe("update TheUser set name=? where id=3", USE_SLAVE, "NewValue");

//强制模式，读主库
TheUser u1 =new TheUser().useContext(ctx).put("id", 3L).load(USE_MASTER);

//强制模式，随机读一个从库
TheUser u2 = user.loadById(TheUser.class, 3L, USE_SLAVE)
}
```

不管DbContext中setMasterSlaveOption设定的是什么策略，不管是否在事务中，强制模式具有最高执行权。

分库分表(Sharding)

jSqlBox支持分库分表，即通常说的Sharding，但jSqlBox并没有做到智能化，如果一个SQL执行出现跨库或跨表现象，它不会象某些Sharding工具那样自动进行复杂的SQL智能分析、SQL分拆执行、查询结果自动汇总，而是简单地抛出一个运行时异常。因为作者认为，一个Sharding出现了跨表甚至跨库访问，本身就是不正常的，业务人员应该尽最大可能避免设计出这样的SQL。如果业务人员很清楚某个sharding操作必须跨库或跨表，是业务要求，无法避免的，那么业务人员肯定也有能力针对这种业务手工进行SQL分拆、汇总操作，甚至进行性能优化。目前一些Sharding工具如Sharding-JDBC，虽然具有SQL自动分拆、汇总功能，但是也有不少问题，如过分复杂，不支持原生SQL、只支持少数数据库, 夺取事务控制权等。对一个基本上不是问题的问题却要付出这些沉重代价，我个人认为没必要, 只需要在DAO层提供一些基本的Sharding手段就足够了。
在DAO层还是在数据库代理层进行Sharding, 各有利弊，有关这方面的比较，还可以参见这篇文章。
退一步来说，即使数据库代理层的sharding工具做的再好，也和jSqlBox这种在DAO层做Sharding的工具没有冲突，因为它们处在不同的级别，一个是驱动层，一个是DAO层，开发者可以自行选择自己喜欢的方式。

SlqBoxContext中与分库分表相关的配置方法有：

```
setMasters(DbContext[]) 设定一组主库
setShardingTools(ShardingTool[]) 设定一组Sharding工具
setSnowflakeCreator(SnowflakeCreator)设定一个分布式主键生成器
```

另外，还有@ShardTable、 @ShardDatabase、@Snowflake三个实体注解，ShardTB和ShardDB两个方法用于SQL中调用，详见下面说明：

分库分表第一步：在实体字段上加注解

```
public static class TheUser extends ActiveRecord {
	@ShardTable({ "MOD", "8" })
	@Snowflake
	@Id
	private Long id;

	@ShardDatabase({ "RANGE", "10" })
	@Id
	private Long databaseId;
	...
```

说明：
@ShardTable({ "MOD", "8" })表示id是一个分表键，分表策略是以8为基准取余，例如当id值为8时，对应数据库表名为theuser_0，当id为9时，对应theuser_1数据库表, 注意只要用到了分表，所有表名都必须是 "表名"+下划线+序号 这种命名格式。
@ShardDabase({ "RANGE", "10" })注解表示databaseId是一个分库键，分库策略是以10为范围，例如当databaseId值为0到9时，存放在第0个数据库， 当值为10到19时，存放在第二个数据库...,以此类推。
@Snowflake标记当前字段的值是一个Snowflake分布式键,为Long类型，它的值将由jSqlBox来填写，见下文。

@ShardTable注解和@ShardDatabase注解可以叠加在同一个列上或单独用，通常建议放在一个唯一的主键列上，有利于简化编程。

分库分表第二步：配置主库数组

配置主库数组，一个库对应一个数据源，也就是对应一个DbContext实例，形成一个数组结构，然后每个DbContext实例都调用setMasters()方法将这个数组赋给自已：

```
DbContext[] masters = new DbContext[MASTER_DATABASE_QTY];
for (int i = 0; i < MASTER_DATABASE_QTY; i++) {
   masters[i] = new DbContext(dataSource[i]);
   masters[i].setMasters(masters);
   masters[i].setSnowflakeCreator(new SnowflakeCreator(5, 5, 0, i));
}

TableModel model = TableModelUtils.entity2Model(TheUser.class);
for (int i = 0; i < MASTER_DATABASE_QTY; i++) {
   for (int j = 0; j < TABLE_QTY; j++) {//顺便把空表都建好
      model.setTableName("TheUser" + "_" + j);//改表名
      for (String ddl : masters[i].toCreateDDL(model))
         masters[i].exe(ddl);//建表
   }
}
DbContext.setGlobalDbContext(masters[0]);
```

数组是有序的，也就是说，如果需要找到5号库,就相当于数组中任一个库调用以下方法： masters[任意].getMasters[5]
上例中顺便把空表都建好了，最后一行是随便选一个主库作为当前库，ActiveRecord操作需要一个缺省库。(见ActiveRecord一节)

每一个主库还可以配置一堆从库，每一个主库和从库都可以创建一堆分表table_0,table_1...，有同学可能问，如果这些主、从库哪一个挂掉了怎么办？jSqlBox不解决这个问题，作为一个轻量级DAO工具来说，它只是提供搭建这样一个结构的最基础的平台，如何做到容错性，必须通过业务方法、二次开发(如自定义具有容错性的主从、分库、分表逻辑)、或将异常传递给上层架构如微服务等来解决。

分库分表第三步：在程序中正常使用CRUD方法，分库分表规则会自动生效

```
for (i:=0;i<100;i++)
 new TheUser().put("databaseId", i).insert();
```

运行一下， TheUser按照第一步中描述的Sharding规则被存到了各个数据库和分表中。

以上是pintea系中的ActiveRecord用法，下面是pintea系中使用SQL方法的例子：

```
masters[2].exe(TheUser.class, "insert into ", shardTB(10), shardDB(3),
		" (id, name, databaseId) values(?,?,?)", param(10, "u1", 3), USE_BOTH, new PrintSqlHandler());
Assert.assertEquals(1, masters[2].qryLongValue(TheUser.class, "select count(*) from ", shardTB(10),
		shardDB(3), USE_SLAVE, new PrintSqlHandler()));
Assert.assertEquals(1,
```

```
            masters[2].qryLongValue(TheUser.class, "select count(*) from ", shardTB(10), shardDB(3)));
masters[2].exe(TheUser.class, "insert into ", shard(3), " (id, name) values(?,?)", param(10, "u1"),
            USE_BOTH, new PrintSqlHandler());
```

ShardTB方法可以根据给出的分表键值和分表策略返回当前实体对应的表名（即一个字符串)。 ShardDB方法可以根据给出的分库键值和分库策略返回当前实体对应的库(即DbContext实例), 当一个DbContext实例出现在Sql方法中时，它会强制夺取SQL执行权，也就达到了分库的目的。
从2.0.4版起，当@ShardTable和@ShardDatabase注记在同一个单主键上时，可以用Shard(主键值)一个方法就行了,它同时返回表名和库实例。 以上三个方法的参数视分库分表策略不同，计算时，参数可以接收Collection、数组、多个参数，但是要注意的是，如果计算出来多个表名或多个库，会直接抛出一个运行时异常，因为jSqlBox不支持自动跨库、跨表汇总SQL结果，这需要程序员手工进行跨库跨表SQL汇总，或尽量在业务设计时避免出现这种操作。

分库分表之配置

现在再回过头来说一下DbContext中的两个配置方法：
setShardingTools(ShardingTool[])
用于设定一组Sharding工具, 对于@ShardTable({ "MOD", "8" })和@ShardDatabase({ "RANGE", "10" })这两个注解，是由jSqlBox中自带默认的ShardingTool接口的实例来处理的。但是jSqlBox只支持最简单的MOD和RANGE这两种策略，如果想加入其它用户自定义的Sharding策略，例如处理@ShardTable({ "Unknow", "8" }) 这种注解，就必须调用setShardingTools方法来配置自己的ShardingTool，它可以接受一个ShardingTool数组，以处理不同的策略，注意setShardingTools方法会清空jSqlBox自带的策略，如果想保留自带的两个策略，必须这样设置：
ctx.setShardingTools(new ShardingModTool(), new ShardingRangeTool(), new 自定义策略1(), new 自定义策略2()...}

setSnowflakeCreator方法设定分布式主键生成器 jSqlBox自带一个分布式主键生成器，名为SnowflakeCreator，它有四个构建参数，含义分别为数据中心位数(0-9),工作站位数(0-9),数据中心ID(0-511)，工作站ID(0-511), 前2个参数的和必须为10,后二个参数的和必须小于1024，更多关于Snowflake算法的了解见这里，因为它标记了工作站的唯一ID，所以每个DbContext的SnowflakeCreator都不能相同，只能在运行期调用以下方法在运行期为每个主库动态配置一个分布式主键生成器：
masters[i].setSnowflakeCreator(new SnowflakeCreator(5, 5, 0, i));
SnowflakeCreator分布式主键相比与UUID的优点在于它是一个long类型的值，如果精心维护好，保证每个应用的机器id不同，而且每台机器的时间不回调，可以做到整个分布式环境内不会出现相同的id。

以上分库分表内容对应的源码在单元测试的ShardingModToolTest.java和ShardingRangeToolTest.java两个文件中。

分库分表之:事务

Sharding中的分表因为所有表都在一个库里，没有事务的困扰。而分库，尤其是多主库的情况下(不是读写分离这种情况)，就可能存在分布式事务这个难题，请详见分布式事务一节。

多租户

从5.0.3版起，jSqlBox添加了支持多租户功能，这个多租户相当于一种特殊的分库，主要特点是可以不依赖于实体定义，而是可以直接根据URL/IP地址/userID等进行运行期动态分库, 它通过在初始化DbContext时设定一个TentantGetter实例来完成，具体示例可以见：这里 实际使用时，要在用户自定义的getTentant方法返回一个可用的DbContext实例，通常是预先设定好的DbContext数组中的一个实例。因为这个多租户功能比较简单，所以就不多介绍了，但是要注意多租户的主DbContext不能作为getTenant方法的返回值，这样会造成循环引用。

18 分布式事务

分布式事务原则：尽量不使用分布式事务

分布式事务如何保证高性能和数据一致性、可用性一直是个难点（参见CAP理论），一个最基本的原则是: 如果可能，尽量不要采用分布式事务设计。例如在jSqlBox中尽量采用TinyTx或GroupTx来操纵单机事务，不引入分布式事务，这两个事务管理器是支持分库分表的，如果业务采用了Sharding方案，实际上往往可以在支持大数据量的前题下，不制造出分布式事务问题，当然这对业务设计、持久层工具的要求比较高，需要把所有相关的业务数据都会保存同一个数据库中，也就是说在分库分表的情况下也会收敛到单个数据源上进行数据库操作。如果采用微服务架构，把所有订单放在一个库里，所有用户放在另一个库里，则编程虽然简单，但却制造出了微服务之间的分布式事务问题，不得不引入分布式事务工具来解决这个问题。 分布式事务工具不是说不可以使用，但是它配置维护复杂，而且一旦硬件出错而需要手工进行数据一致性修复工作时，对运维人员的技术水平要求会非常高。

分布式事务之XA事务

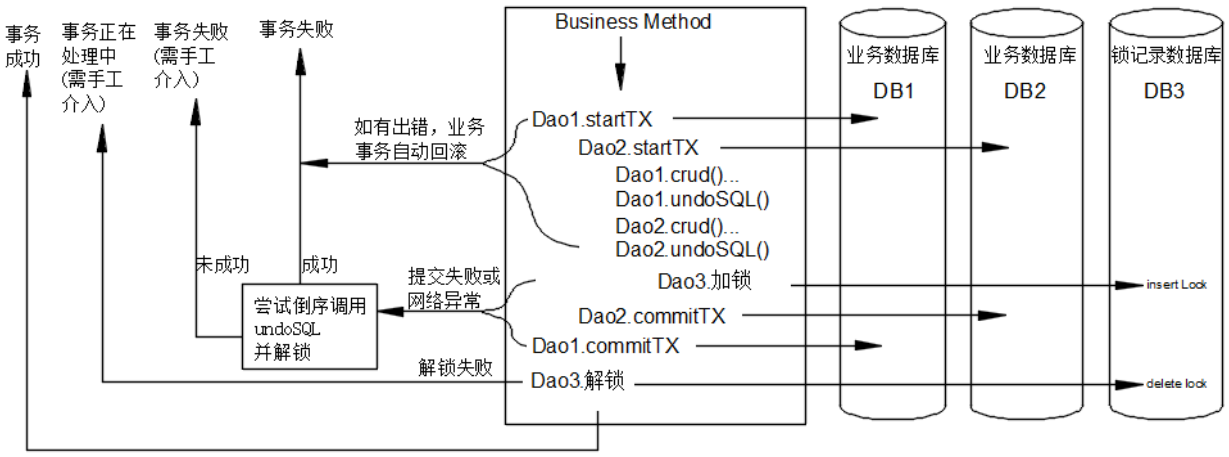
分布式事务一种方案是采用XA事务，它利用数据库本身对XA分布式事务协议的支持完成。
在demo\jsqlbox-atomikos目录下，有一个分布式事务的演示，主要利用到了Atomikos和Spring对XA事务的支持，没有发明什么新东西，只是简单地演示了一下jSqlBox的分库分表结合分布式XA事务的使用。XA事务基于两阶段提交，需要数据库本身支持，如果并发量大时，锁表冲突，可能造成性能问题。 本人不推荐使用XA事务，不光是因为它的性能差，而且本质上它的理论基础是有问题的，例如在两阶段提交或3阶段提交的最后步骤网络中断，还是有可能造成数据不一致的。

分布式事务之 Seata(原Fescar)事务

Seata是一个采用自动补偿方案的开源分布式事务工具，这是阿里开源出来的项目，非常火热，它的特点是对业务入侵小，通过分析SQL来自动创建回滚SQL。 但个人感觉它基于分析SQL语法来创建回滚SQL(这个和Sharding-JDBC有点类似，也是建立在分析SQL语法基础上)，导致整个工具的性能、可维护性、数据库兼容性都比较差，到目前为止，感觉还没有完全成熟。 jSqlBox目前没有对Seata事务的支持，以后可能会引入。

分布式事务之jSqlBox的Gtx事务

Gtx事务是jSqlBox3.0.0自带的分布式事务模块，这使得它可能是第一款自带分布式事务的DAO工具。它的总体思路和Seata类似，也是通过生成反向记录来自动回滚，减小对业务的侵入，但jSqlBox采用的思路是将分布式事务建立在ORM工具之上，不是分析SQL内容，而是记录实体的插入、删除、修改操作，以生成回滚记录。这样一来，在实现难度上就降低了一个等级，以牺牲SQL支持功能达到最好的数据库兼容性，支持所有数据库。在具体实现上，它通过采用最大保证完成模式结合全局记录锁的方案，架构请参见Bag分布式事务对SAGA分布式事务的改进一文。



它采用事务嵌套模式，只要内层的事务提交失败，就会引发包含它的外层事务回滚。因为网络中断等原因没有提交或回滚的事务则由一个守护服务来定时回滚事务和解锁被锁定的资源。

以下为一个分布式事务的演示:

```
public class GtxTest {
 DbContext[] ctx = new DbContext[3];

 private static DataSource newTestDataSource() {
  HikariDataSource ds = new HikariDataSource();
  ds.setDriverClassName("org.h2.Driver");
  ds.setJdbcUrl("jdbc:h2:mem:" + new Random().nextLong() // random h2 ds name
    + ";MODE=MYSQL;DB_CLOSE_DELAY=-1;TRACE_LEVEL_SYSTEM_OUT=0");
```

```
 ds.setUsername("sa");
 ds.setPassword("");
 return ds;
}

@Before
public void init() {
 DbContextlock = new DbContext(newTestDataSource());
 lock.setName("lock");
 lock.executeDDL(lock.toCreateDDL(GtxId.class));
 lock.executeDDL(lock.toCreateDDL(GtxLock.class));
 lock.executeDDL(lock.toCreateGtxLogDDL(Usr.class));
 GtxConnectionManager lockCM = new GtxConnectionManager(lock);
 for (int i = 0; i < 3; i++) {
  ctx[i] = new DbContext(newTestDataSource());
  ctx[i].setName("db");
  ctx[i].setDbCode(i);
  ctx[i].setConnectionManager(lockCM);
  ctx[i].setMasters(ctx);
  ctx[i].executeDDL(ctx[i].toCreateDDL(GtxTag.class));
  ctx[i].executeDDL(ctx[i].toCreateDDL(Usr.class));
 }
}

public void Div0Test() {
 ctx[0].startTrans();//任一个被lockCM登记的ctx都可以启动分布式事务
 try {
  new Usr().insert(ctx[0]);//这四行分别在三个数据库插入新记录
  new Usr().insert(ctx[1]);
  new Usr().insert(ctx[1]);
  new Usr().insert(ctx[2]);
  System.out.println(1 / 0);//强制出错
  ctx[0].commitTrans();
 } catch (Exception e) {
  TxResult result=ctx[0].rollbackTrans();//这里会回滚所有三个数据库事务
  GtxUnlockServ.forceUnlock(ctx[0], result); //这行不需要，只是为了单元测试
 }
 Assert.assertEquals(0, ctx[0].entityCountAll(Usr.class));
 Assert.assertEquals(0, ctx[1].entityCountAll(Usr.class));
 Assert.assertEquals(0, ctx[2].entityCountAll(Usr.class));
}
}
```

上例是最简单的一个分布式事务演示，GTX事务如果在事务提交中出错，无论有无部分事务提交发生，最终数据一致性都能保证。
注意Gtx只支持针对单个实体进行的存取操作，如insert/update/delete/load方法，不支持批量查询或更新的方法如findAll方法，如果利用批量查询加载的实体需要进行分布式事务管理，必须用load方法(或exist/existStrict方法检查)重新加载一次，以保证生成回滚记录并创建全局事务锁。

注意上例forceUnlock方法仅用于单元测试，实际生产项目中应该去掉GtxUnlockServ.forceUnlock(ctx[0], result)这一行，而改成用GtxUnlockServ.start(ctx, loopInterval，maxLoopQty);方法单独在LockServ上开启一个轮循解锁服务，第二个参数为解锁间隔，单位为秒，必须设置成一个远大于数据库事务超时时间的值比如500秒，第三个参数可以设为0，表示没有最大解锁次数限制。运维人员可以通过监控LockServ上的记录有没有被定时清除来检查是否有需要手工介入进行解锁处理的分布式事务，这通常是因为硬件损坏、编程错误或网络永久性中断才会出现的故障。
一般来说短暂的网络中断不会造成需要手工介入的问题，因为轮循解锁服务会在网络恢复后自动解锁并提交或回滚事务。当需要手工介入进行解锁处理时，相同的事务不能再入，也就是说涉及到的表格行被锁定，只能读取但不能被第二个分布式事务更改内容。

GTX是一个支持分库、分表，支持锁服务器本身分库的分布式事务工具，见下例：

```
public class DemoUsr extends ActiveRecord<DemoUsr> {
 @Id
 @ShardDatabase({ "MOD", "3" })
 Integer id;

 @ShardTable({ "RANGE", "10" })
 Integer age;
//Getter & Setter 略
}


public class GtxShardDbTbLockDbTest {
 private static final int DB_SHARD_QTY = 5;
```

```
DbContext[] ctxs = new DbContext[3];
DbContext[] lockCtxs = new DbContext[3];

private static DataSource newTestDataSource() {
 //同上略
}

@Before
public void init() {
 DbContext.resetGlobalVariants();
 DbContext.setGlobalNextAllowShowSql(true);

 for (int i = 0; i < 3; i++) {
  lockCtxs[i] = new DbContext(newTestDataSource());
  lockCtxs[i].setName("lock");
  lockCtxs[i].setDbCode(i);
  lockCtxs[i].executeDDL(lockCtxs[i].toCreateDDL(GtxId.class));
  lockCtxs[i].executeDDL(lockCtxs[i].toCreateDDL(GtxLock.class));
  lockCtxs[i].executeDDL(lockCtxs[i].toCreateGtxLogDDL(DemoUsr.class));
  lockCtxs[i].setMasters(lockCtxs);
 }

 GtxConnectionManager lockCM = new GtxConnectionManager(lockCtxs[0]);// random choose 1
 for (int i = 0; i < 3; i++) {
  ctxs[i] = new DbContext(newTestDataSource());
  ctxs[i].setName("db");
  ctxs[i].setDbCode(i);
  ctxs[i].setConnectionManager(lockCM);
  ctxs[i].setMasters(ctxs);
  ctxs[i].executeDDL(ctxs[i].toCreateDDL(GtxTag.class));
  TableModel model = TableModelUtils.entity2Model(DemoUsr.class);
  for (int j = 0; j < DB_SHARD_QTY; j++) {
   model.setTableName("DemoUsr_" + j);
   ctxs[i].executeDDL(ctxs[i].toCreateDDL(model));
  }
 }
 DbContext.setGlobalDbContext(ctxs[0]);// the default ctx
}

@Test
public void commitFailTest() {
 ctxs[0].startTransOnLockDb(1);
 try {
  new DemoUsr().setId(0).setAge(0).insert(); // locker1, db0, tb0
  new DemoUsr().setId(1).setAge(10).insert(); // locker1, db1, tb1
  new DemoUsr().setId(4).setAge(11).insert(); // locker1, db1, tb1
  new DemoUsr().setId(2).setAge(40).insert(); // locker1, db2, tb4
  Assert.assertEquals(1, ctxs[0].qryIntValue("select count(1) from DemoUsr_0"));
  Assert.assertEquals(2, ctxs[1].qryIntValue("select count(1) from DemoUsr_1"));
  Assert.assertEquals(1, ctxs[2].qryIntValue("select count(1) from DemoUsr_4"));
  ctxs[2].setForceCommitFail(); //模拟DB2事务提交出错
  ctxs[0].commitTrans();
 } catch (Exception e) {
  TxResult result = ctxs[0].rollbackTrans();
  GtxUnlockServ.forceUnlock(1, ctxs[0], result);//单元测试强制解锁1号服务器
 }
 Assert.assertEquals(0, ctxs[0].qryIntValue("select count(1) from DemoUsr_0"));
 Assert.assertEquals(0, ctxs[1].qryIntValue("select count(1) from DemoUsr_1"));
 Assert.assertEquals(0, ctxs[2].qryIntValue("select count(1) from DemoUsr_4"));
}

}
```

这个例子中，DamoUser会根据id分库键和age键分别存放到不同的数据库和分表里，另外在开启分布式事务时，可以手工指定当前锁服务器的序号，这样可以通过配置一个锁服务器集群来提高事务处理性能，当然这种情况下，锁服务器的指定通常是与业务有相关的，例如红包转账分布式事务，可以红包的ID取模作为锁服务器的序号。
实际使用时，GtxUnlockServ.forceUnlock(1, ctxs[0], result);不再出现，而是单独用GtxUnlockServ.start(ctx, loopInterval，maxLoopQty)方法来开启一个解锁服务。
更多关于分布式事务的使用请参见单元测试目录jsqlbox/function/gtx下的几个测试示例。

最后补充一下，以上分布式事务成立的前提是锁服务器本身不崩溃，锁记录不会丢失，这点很关键，jSqlBox相当于将多机的可靠性转移到了单机的可靠性上，而维护单机可靠性是一个相对容易、成熟的技术，对于高可靠性要求的场合，可以用Poxas协议搭建数据库集群来保证锁服务器本身的可靠性。

19 批处理

如果有大批的插入、更新语句，可以利用数据库的批量操作功能以加快插入、更新速度，前提是数据库要支持批量操作功能(如MySQL在jdbcURL中要设定rewriteBatchedStatements=true)。在jSqlBox中，批处理有以下几种用法：

1.继承于DbUtils的批处理方法：

```
batch(Connection, String, Object[][])
batch(String, Object[][])
insertBatch(Connection, String, ResultSetHandler<T>, Object[][])
insertBatch(String, ResultSetHandler<T>, Object[][])
```

使用这些方法必须捕获SqlExecption异常，参数是二维数组。

2.从jDbPro模块开如添加的批处理方法：

```
nBatch(Connection, String, List<Object[]>)
nBatch(String, List<Object[]>)
nInsertBatch(Connection, String, ResultSetHandler<T>, List<Object[]>)
nInsertBatch(String, ResultSetHandler<T>, List<Object[]>)
```

这些n开头的方法表示不会抛出必须捕捉的SqlExecption异常，而是抛出运行期异常。这些方法的参数是List<Object[]>。

3.批处理开关方法

有以下几个方法：

```
nBatchBegin()   打开批处理开关
nBatchEnd()     结束并刷新未存盘的批处理操作
nBatchFlush()   手工刷新未存盘的批处理操作
isBatchEnabled() 获取当前批处理开关状态
```

对除DbUtils自带的方法外所有的SQL插入、更新操作(即不抛出SqlException的upd/ins/exe方法)，可以用DbContext实例的nBatchBegin()方法设定一个开关, 批处理开始前调用ctx.nBatchBegin()方法，结束后调用ctx.nBatchEnd方法，则所有插入、更新方法将自动汇总成批量操作。系统默认每隔300次SQL批量执行后自动调用一次nBatchFlush()方法，也可以手工调用nBatchFlush()方法强制将缓存的批处理操作刷新到数据库。

使用批处理开关方法要注意:

1. 批处理开关方法必须用try... finally块确保在最后调用nBatchEnd()方法，否则批处理开关一直打开着，会造成后续程序逻辑错误。
2. 批处理开关方法只支持jSqlBox的SQL方法（即不抛出SqlException的upd/ins/exe/entityXxx方法，以及ActiveRecord的CRUD方法)，不支持DbUtils自带的要捕捉SqlException的方法，如execute、update、insert等。
3. 批处理开关方法仅适用于插入、更新操作，中间不应有读操作，因为插入、更新操作被缓存起来批量执行，在这过程中读操作可能会读到脏数据。所以批处理开关方法建议仅用于数据导入、测试等特殊场合，日常业务操作不建议使用批处理开关方法。

以下为各种批处理方式在MySql上插入300万条记录的测试(i7 CPU)，源码详见单元测试下的BatchTest.java:

```
nBatchBegin/nBatchEnd execute 3000000 SQLs time used: 11.606 s
nBatch(Sql, List<Object[]>) method execute 3000000 SQLs time used:   6.41 s
batch(Sql, Object[][]) method execute 3000000 SQLs time used:  5.836 s
```

实测在小数据量时，批处理开关方法反而要快一些，但是在插入数据量超过5万条记录时，批处理开关方法的插入速度要比DbUtils自带的batch方法慢一倍左右。个人猜测可能与内存碎片造成了垃圾回收有关。

20 查询缓存和缓存翻译

查询缓存器

jSqlBox自带一个查询缓存器SimpleCacheHandler， 它是SqlHandler拦截器的一个实现。它只针对查询方法生效，第一次查询时读取数据库，后续的查询将会从缓存中读取。使用SimpleCacheHandler查询缓存，通常配置成单例或全局变量，在pintea系的各种查询方法中作为参数传入:

```
SimpleCacheHandler simpleCache=new SimpleCacheHandler();
//第一次发出SQL到数据库
qry(simpleCache, new EntityListHandler(), DemoUser.class, "select * from DemoUser where age>?", par(0));
//第二次将从缓存中读，如果SQL和参数一样的话。
qry(simpleCache, new EntityListHandler(), DemoUser.class, "select * from DemoUser where age>?", par(0));
```

以上测试代码位于单元测试的SqlHandlersTest.java。

jSqlBox自带的缓存器SimpleCacheHandler是个简单的LRU缓存，利用内存中的LinkedHashMap表来缓存SQL查询结果，当缓存存满数据时，会把最久没有被访问到的数据清除，SimpleCacheHandler有两种构造方式：

```
SimpleCacheHandler()
SimpleCacheHandler(int capacity, int aliveSeconds)
```

后一种构造方式允许设定缓存大小和缓存失效时间，缓存失效时间单位是秒（内部会向上取值调整成与1、10、100、1000、10000...最接近的值)。 如果采用第一种无参构造器，默认缓存大小是500条查询记录和1000秒缓存失效时间。

使用SimpleCacheHandler查询缓存器的另一种方式是在DbContext 构造时将缓存拦截器SimpleCacheHandler设置成全局拦截器(见jSqlBox配置一节)，如:

```
DbContext ctx=new DbContext (ds);
ctx.setHandlers(new SimpleCacheHandler());
```

这样就不必每次在查询方法中手工加入缓存拦截器了，但是因为缓存有可能造成脏数据的存在，这种方式通常不应该参与事务操作，可以考虑单独创建一个配置了缓存的DbContext，给它分配专门的数据源，仅用于查询用。

jSqlBox自带的查询缓存机制比较简单，与MyBatis一样，它是粗粒度的，不是针对行集的，缓存的主键采用"时间+SQL+参数"这种方式, 只要SQL拼写或参数值有任意一点不同，jSqlBox即认为这是两个不同的查询，因此哪些查询需要使用缓存需要仔细考虑，参数经常变化的查询不应该使用缓存。

如果想要编写自己的缓存类，可以参考"Handler类介绍"一节和SimpleCacheHandler类的源码。

缓存翻译

缓存翻译(有时被称为“数据字典”)可以用内存中缓存的数据库表来避免多表关联查询，简化SQL的书写，并提高查询效率。以下是一个使用了缓存翻译的示例，完整示例代码详见单元测试中的CacheTranslateTest.java：

```
Map<Integer, Map<String, Object>> users = ctx.qry("select * from users", new KeyedHandler<Integer>("id"));
Map<Integer, Map<String, Object>> groups = ctx.qry("select * from groups", new KeyedHandler<Integer>("id"));
List<Map<String, Object>> orders = ctx.qryMapList("select id,orderNo,userId,groupId from orders where id>'10' ");
CacheTransUtils.translate(orders, users, "userID", "name", "userName", "age", "userAge", groups, "groupId", "groupName", "groupName");
```

以上translate方法对orders这个List集合，根据缓存users和groups表，添加了userName、userAge和groupName三个列。
translate方法的参数个数不限，但是必须严格按照以下参数顺序来使用： translate(要处理的集合，缓存A,"集合中的列名","缓存A中的列名1"，"列名1别名","缓存A中的列名2"，"列名2别名"....., 缓存B,"集合中的列名","缓存B中的列名1"，"列名1别名"...)

注意这个缓存翻译功能是针对List<Map<String,Object>>这个数据结构的，是对SQL的查询结果进行后处理，它与DAO工具实际上关系不大，也就是说，其它的DAO工具如MyBatis,JdbcTemplate等也可以调用jSqlBox的这个CacheTransUtils.translate方法实现缓存翻译功能。

缓存翻译通常缓存不常变动的表格(即数据字典)到内存中，如果数据字典偶尔有变动，可以有两种方案，一种是在更改数据字典时，强制刷新缓存。另一种方案是每次使用缓存翻译功能时，都重新查询数据字典，在查询数据字典给它传递一个缓存拦截器，由缓存拦截器的失效时间来决定是否实际发出SQL到数据库查询。

jSqlBox缓存翻译的缺点：被翻译后添加的这些字段，不能出现在SQL查询条件中，这是它的一个使用限制。如果即想将这些字段作为SQL参数参与查询，又不想采用多表关联这种传统写法，可以考虑打破数据库范式，采用添加冗余字段的技巧。

21 固定和动态配置

jSqlBox同时支持固定配置和动态配置，这是它的一个特点。

先看一下固定配置:

```
@Table(name = "emailtb")
public class Email{
 @Id
 String id;
 String emailName;

 @SingleFKey(refs = { "usertb", "id" })
 String userId;
       ......
}
```

jSqlBox支持对象关联查询，这意味着对于一个实体Bean如上面的Email类来说，它必须知道这个类对应哪一个数据库表，以及每个实体属性对应数据库表的哪一个列，以及各个列与其它表格之间的外键约束(对jSqlBox来说可以利用外键来进行关联查询，详见实体关联查询一节），通过Annotation注解，可以达到这个目的。但是这种注解方式的配置，是很难在运行期改变的，笔者称其为固定配置，还有Hibernate和MyBatis中的XML配置，在运行期是很难改变的，所以也是一种固定配置。固定配置的优点是稳定，程序中只需要配置一次，缺点是不灵活，例如Hibernate的配置只能对一个实体有一种配置，当实体间关联关系不确定时或需要临时改变时就没有办法了。MyBatis的对策是对应不同的查询，创建多份不同的XML固定配置，带来的后果是每做一个复杂的关联查询就要建一个XML条目，其中的字段名、属性名等大量重复，增加了不必要的工作量，提高了维护成本。

当持久层工具将注解或XML配置的内容读入内存并解析后，生成Java对象，并公开它的存取方法，允许在运行期创建或改变配置，本人称之为动态配置，目前很少有持久层工具做到这点。

jSqlBox同时支持固定配置和动态配置

利用固定配置打下整个项目配置的基础，对于需要临时改变配置的场合下，可以在运行期临时创建或改变固定配置，这种改变相比与从头到尾创建一个新的配置来说，工作量很少, 因此jSqlBox同时具有了固定配置和动态配置的优点。

jSqlBox的固定和动态配置功能，是由jDialects模块来承担，这是一个可以独立使用的数据库建模工具，可以根据Bean注解或Java链式方法创建一个与具体数据库无关的虚拟表模型，这个模型也被jSqlBox利用来作为配置对象，可以在运行期修改，而且修改它的Java方法和建立它的方法是一样的，因为本来就是同一个对象。

以下是固定配置和动态配置结合使用的一个示例：

```
public static class UserDemo extends ActiveRecord<UserDemo> {
      private String id;

      @Column(name = "user_name2", length = 32)
      private String userName;

      public String getUserName() {  return userName; }

      public void setUserName(String userName) { this.userName = userName; }

      public String getId() {return id; }

      public void setId(String id) { this.id = id; }

      public static void config(TableModel t) {
         t.setTableName("table2");
         t.column("user_name2").setColumnName("user_name3");
      }
   }

//测试方法
   public void testDynamicConfig() {
      TableModel model = TableModelUtils.entity2Model(UserDemo.class);
      model.column("id").pkey();
      createAndRegTables(model);

      UserDemo u1 = new UserDemo();
      u1.setId("u1");
      u1.setUserName("Tom");
```

```
        u1.insert(model);

        UserDemo u2 = ctx.entityLoadById(UserDemo.class, "u1", model);
        Assert.assertEquals("Tom", u2.getUserName());

        model.column("userName").setTransientable(true);
        UserDemo u3 = ctx.entityLoadById(UserDemo.class, "u1", model);
        Assert.assertEquals(null, u3.getUserName());
    }
```

说明:

- TableModelUtils.entity2Model(UserDemo.class)方法获取针对实体的配置的一个副本，可以修改。
- config方法是针对实体类的固定配置方法(这是jDialects中的一个约定)。
- t.column("user_name2").setColumnName("user_name3")方法改变映射列名为"user_name3"，如果要获取一个列对象，可以用column(列名)或column(实体属性名)来获取，效果一样并且不区分大小写。
- model.column("id").pkey() 方法动态配置了User.class实体类的主键例，如果没有这条语句，u1.insert(model)会出错，因为主键列没有。
- 注意CRUD方法中的model可选参数，它是一个TableModel类型实例，如果一个pintea系方法中出现TableModel实例作为参数，则它的配置将会覆盖掉对应实体类的缺省固定配置。
- model.column("userName").setTransientable(true)动态配置了userName属性为"Transientable"类型，于是eLoadById(UserDemo.class, "u1", model)方法将会在加载UserDemo实体时忽略掉userName字段。

本节的示例源码位于单元测试目录下的DynamicConfigTest.java。

对第三方POJO进行配置

在没有源码的情况下，jSqlBox也可以对第三方POJO进行配置，以下是一个示例。

```
    @Test
    public void doPojoConfigTest() {
        TableModel model = TableModelUtils.entity2Model(PojoDemo.class);
        model.column("id").pkey().uuid32();
        TableModelUtils.bindGlobalModel(PojoDemo.class, model);

        PojoDemo pojo = new PojoDemo();
        pojo.setName("Tom");
        ctx.eInsert(pojo);
        ctx.eDelete(pojo);
    }
```

这里假设PojoDemo是一个第三方纯POJO类，它有id和name两个字符串实体属性，并且没有任何注解，jSqlBox利用动态配置，强行给它设定id为UUID32类型，并设为主键。这里用到了jDialects中的TableModelUtils工具类，entity2Model生成一个类的TableModel配置，bindGlobalModel绑定一个类的配置，从方法命名上可以看出，这是一个全局绑定，也就是说今后jSqlBox中所有CRUD操作这个POJO时都会读取它这个绑定的全局配置，而不需要每次将配置当作参数传递给CRUD方法中去。

演示项目

在SpringBoot中使用

这是一个为了演示jSqlBox在springboot环境中使用而创建的项目，位于demo目录下。项目架构是SpringBoot+jSqlBox，用到了Spring的IOC/AOP、声明式事务、MVC模块。编译及运行本项目需Java8或以上环境。

DAO工具与Web开发环境集成，主要就是事务的配置。
jSqlBox在SpringBoot配置非常简单，首先pom.xml里先添加jSqlBox的库依赖：

```
    <dependency>
      <groupId>com.github.drinkjava2</groupId>
      <artifactId>jsqlbox</artifactId>
      <version>5.0.3.jre8</version> <!--或最新版-->
    </dependency>
```

jSqlBox没有什么starter之类的发布包，需要在Spring里手工添加一个Bean单例配置，配置比较简单，只有4行代码，但是新手可能看不明白，因为Spring建立在约定的基础上，屏蔽了复杂性，其具体原理可参见jSqlBox的事务配置一节介绍。

```
@SpringBootApplication
public class JsqlboxInSpringbootApplication {
    @Autowired
    DataSource ds;

    public static void main(String[] args) {
        SpringApplication.run(JsqlboxInSpringbootApplication.class, args);
    }

    @Bean
    public DbContext createDefaultDbContext () {
        DbContext ctx = new DbContext(ds);
        ctx.setConnectionManager(SpringTxConnectionManager.instance());
        DbContext .setGlobalDbContext(ctx);// 设定静态全局上下文
        return ctx;
    }
}
```

配置完成后，就可以在server类上用Spring的@Transactional注解标注，享用Spring提供的声明式事务了，当然,service类实例(通常是单例)必须从Spring环境获取。

项目编译及运行：mvn spring-boot:run
打开浏览器查看: http://localhost

在ActFramework中使用

本演示项目位于demo目录下的jsqlbox-actframework, 这个项目用来演示jSqlBox在ActFramework中的配置和使用。

Dao工具最主要的配置就是事务，这个项目里同时演示了使用通用IOC/AOP工具Guice和jBeanBox来进行声明式事务的配置，也就是说没有使用ActFramework的DI和AOP功能。除了Web部分使用了Actframework,Dao部分是独立的。因为配置比较多，请自行打开源码查看。涉及的事务配置原理请参见jSqlBox “事务配置”一节的介绍。

主程序中有两个dao，分别从Guice和jBeanBox环境中获取，这两个dao都具备事务出错自动回滚功能:

```
public Result transfer(int amount, boolean btnA2B, boolean btnB2A, ActionContext context) {
        Account dao = INJECTOR.getInstance(Account.class); // 使用Guice来创建AOP代理类
        Account dao2 = JBEANBOX.getInstance(Account.class); // 使用jBeanBox来创建AOP代理类
        try {
            if (btnA2B)
                dao.transfer(amount, ACC_A, ACC_B);
            else
                dao2.transfer(amount, ACC_B, ACC_A);
            context.flash().success("Transaction committed successfully");
        } catch (Exception e) {
            context.flash().error("Transaction failed. Possible reason: no enough balance in the credit account");
            e.printStackTrace();
        }
        return redirect("/");
    }
```

顺便跑个题：通常AOP工具总是提供依赖注入(IOC)功能的，因为获取Service单例时，程序员不知道Service类是否有事务(注解)配置，不能决定用getEnhanced(Service.class)还是用new service()的方式去生成实例，只能依靠IOC工具的getinstance方法或自动注射去获取，由IOC工具来检测并自行决定是生成一个代理类还是一个普通类。反之，IOC工具却不一定提供AOP功能。

项目编译及运行: mvn clean compile exec:exec
浏览器下查看： http://localhost

另外,这个示例在Windows下运行可能出现DOS窗口关闭后, 后台的Web服务不能自动退出，并占用Web端口，必须在windows下打开任务管理器手工杀掉一个Java.exe进程，这与ActFramework的Web服务有关，已向作者提交issue#849。

在jFinal中使用

演示项目位于demo目录下的jsqlbox-in-jfinal，演示jSqlBox与jFinal的集成使用。jFinal的DAO部分有一个问题是它基于Map的实体类不是一个标准的JavaBean，不利于与其它工具集成使用。这个演示项目里用jSqlBox替换掉它的DAO部分，但是依然使用它的事务及Web功能。

配置的主体部分位于项目的DemoConfig.java文件中，可以自行打开查看，实际上也就是三行代码：

```
public class DemoConfig extends JFinalConfig {

    @Override
    public void afterJFinalStart() {
        DbContext ctx = new DbContext(DbKit.getConfig().getDataSource());
        ctx.setConnectionManager(JFinalTxConnectionManager.instance());
        DbContext.setGlobalDbContext (ctx);
    }
```

声明式事务有3个关键:

1. IOC/AOP工具， 这里就用jFinal自带的。
2. 声明式事务切面处理器， 这里就用jFinal自带的。
3. 连接管理器。不同的声明式事务工具获取和关闭connection的工具类不一样，在jTransactions(jSqlBox内含)模块中为纯JDBC、jFinal、Spring等都准备了一个连接管理器,这里设成JFinalTxConnectionManager.instance()，这是一个单例。

和SpringBoot一样，因为jFinal也自带声明式事务功能，所以jSqlBox配置起来也是轻松加愉快，只要注意它的数据源是从DbKit的Config里取出来的，连接管理器设成JFinalTxConnectionManager就行了，不需要自已去考虑定义事务AOP注解了。
当然，如果有人不想使用jFinal的事务功能也是可以的，因为jSqlBox本身也自带声明式事务模块，可以参考jBooot或jSqlBox-Actframework中的配置，自行配置出自已的IOC/AOP/声明式事务。jFinal的声明式事务@Before(Tx.class)这种写法比较啰嗦，如果使用Spring、Guice或jBeanBox之类的IOC/AOP工具，都可以自定义事务注解，而且不需要加参数，如@Transactional、@MyTX这种写法。

jFinal的自动生成实体源码的功能，在jSqlBox中可以用jDialects的自动生成实体源码功能代替，详见jDialects项目，jSqlBox不鼓励让实体类继承于一个基类，因为文件数多了一个，没必要。代码生成器在项目后期作用不大，例如实体类字段名变更时，还是必须找到实体类源码，手工在IDE里重构，代码生成器帮不上忙。

jFinal的Db+Record模式，在jSqlBox里可以用相似的Tail功能代替，但是不要滥用，因为这种方式不支持重构，不利于项目维护。Tail功能在查询时用到较多，因为查询结果返回的列可能只用一次，没必要在JavaBean里定义对应的字段。

项目启动步骤

1. 启动jFinal: 双击批处理文件 "maven start jfinal.bat"
2. 在浏览器查看效果： http://localhost

注意演示数据的第一行删不掉，目的是为了演示jFinal的声明式事务，删除第一行会抛出一个Div/0错误，事务会自动回滚。

如果要导入Eclipse中运行，必须：
1）命令行下运行maven_eclipse:eclipse，生成Eclipse的配置文件.classpath和.project，导入Eclipse
2) 修改源代码中DemoConfig.java中的path="webapp" 为path="target/jsqlbox-in-jfinal-1.0"
3）运行"maven_clean_package.bat"批处理文件，重新编译项目(以后每次更改源码后都需要)
4) 选中DemoConfig.java，选择“Run as” ->"Java Application"

注意： Tomcat下运行项目需要先删除jetty-server-xxx.jar或改变pom.xml中此项scope为"privided",否则可能会有冲突。
本演示项目需Java1.8或更高版本。

jBooox演示项目

这是一个WebApp演示项目，位于demo目录下，主要架构基于 jbeanbox + jwebbox + jsqlbox，因为三个开源项目都以Box结尾, 所以项目简称jBooox。

jBooox项目的架构特点是各个模块之间完全独立，每个模块都可以单独抽取出来使用，在Maven中央库都有发布。这是它与jFinal、Nutz之类提供一篮子整体解决方案工具的最大区别，jBooox不保证每个模块都令人满意，所以不会将这些模块强行捆绑打包提供。jBooox这种模块式架构设计与Spring系列工具类似，但是代码更短小、架构更松散，每个子模块，甚至连IOC内核、声明式事务这些模块都不是主角，而是随时都可以被替换掉。

jBooox项目各模块简介:

jSqlBox是一个持久层工具，支持多种SQL写法。
jBeanBox是一个IOC/AOP工具, 用于在项目中代替Spring-IOC内核。
jWebBox是一个后端页面布局工具，用于替代Apache-Tiles的布局功能，在本项目中客串充当MVC中的Model和View角色，相当于SpringMVC中的ModelAndView。对于Web编程来说，如果不想引入SpringMVC、Struts、jFinal等比较大的Web框架，jWebBox是一个很好的起点，因为它的源码很少，只有500行。

编译及运行本项目需Java8, 发布到Tomcat7以上或WebLogic环境中去运行。

Spring演示项目

这是一个为了演示jSqlBox在Spring环境中的配置和使用而创建的Web项目，位于demo目录下。项目架构来源于https://github.com/Fruzenshtein/spr-mvc-hib演示项目，原项目架构是Hibernate + SpringTx + SpringIOC + SpringMVC + MySql，本项目将其中的Hibernate用jSqlBox替换掉，MySql用H2内存数据库替换掉(以避免配置数据库的麻烦)，其余部分不变。

这是一个传统Web项目，必须打包war发布到独立的Tomcat等web服务器下运行，已有点过时，推荐使用内嵌Web服务模式，见jsqlbox-springboot示例。但因为SpringBoot的配置极简，看不出什么所以然，所以在这个传统风格的演示项目里反而可以更新楚地看出事务的配置。

编译及运行此项目需本机安装Java8、 Tomcat7或以上版本。

项目中有一个deploy_tomcat.bat批处理文件，请修改它，发布路径指向本机的Tomcat或其它Web服务路径，然后运行它即可发布运行。

与MyBatis混用

MyBatis是国内使用较多的一款DAO工具，但是它的缺点也很明显：XML配置或@Sql注解方式使用起来比较繁琐，没有提供CRUD方法，开发效率低。国产的MyBatis-Plus插件在某些程度上补上了它的一些功能缺失如ActiveRecord功能，但是它的一些固有问题如XML配置和SQL注解繁琐是不能通过插件方式改变的，所以在开发效率上依然是有疑问的。

jSqlBox作为一个全功能持久层工具来说，整个项目只需要jSqlBox就够了，但是对于一些必须使用MyBatis的场合(例如遗留项目)，也可以将jSqlBox与MyBatis的混合起来使用，利用jSqlBox具有的功能如DDL生成、分页、主键生成、跨数据库的函数转换、ActiveRecord、分库分表等弥补MyBatis的短处，提高开发效率。

在demo/jsqlbox-mybatis目录下，演示了jSqlBox和MyBatis在springboot环境中的混搭使用，项目架构是SpringBoot(用到了它的数据源注入、声明式事务、IOC/AOP、MVC) + jSqlBox + MyBatis。

jSqlBox和MyBatis在SpringBoot下混搭配置很简单 首先pom.xml配置里先添加MyBatis-spring-boot-starter和jSqlBox的库依赖：

```
    <dependency>
      <groupId>org.mybatis.spring.boot</groupId>
      <artifactId>mybatis-spring-boot-starter</artifactId>
      <version>1.3.2</version> <!--或最新版-->
    </dependency>

    <dependency>
     <groupId>com.github.drinkjava2</groupId>
     <artifactId>jsqlbox</artifactId>
     <version>5.0.3.jre8</version> <!--或最新版-->
    </dependency>
```

jSqlBox没有什么SpringBoot starter之类的发布包，需要在Spring里手工添加一个Bean单例配置，和前面演示项目"jsqlbox-in-springboot"一样，配置依然是只有几行代码，这是因为SpringBoot的约定造成了MyBatis和jSqlBox使用的是同一个自动注入的数据源，以及同一个Connection连接管理器：

```
@SpringBootApplication
public class JsqlboxInSpringbootApplication {
   @Autowired
   DataSource ds;

   public static void main(String[] args) {
      SpringApplication.run(JsqlboxInSpringbootApplication.class, args);
   }

   @Bean
   public DBContext createDefaultDBContext() {
      DBContext ctx = new DBContext(ds);
      ctx.setConnectionManager(SpringTxConnectionManager.instance());
      DBContext.setGlobalDBContext(ctx);// 设定静态全局上下文
      return ctx;
   }
}
```

然后在程序里就可以在同一个事务下同时使用MyBatis和jSqlBox了，任一个DAO工具出错，另一个DAO工具已提交的数据也会回滚。

编译及运行本项目需Java8或以上环境。

编译及运行：mvn spring-boot:run
在浏览器查看: http://localhost

注：另一种jSqlBox与MyBatis混用的方式是使用MyFat插件对MyBatis增强，详见MyFat项目。

附录1：性能测试

各种不同SQL写法的性能测试

以下是jSqlBox不同SQL写法进行循环十万次CRUD操作的性能测试, 测试源码见单元测试目录下的UsageAndSpeedTest.java，在H2内存数据库上跑(i7 CPU)，可以排除磁盘读写带来的影响，反映出框架本身的性能。

```
Speed test, compare method execute time for repeat 100000 times:
                    pureJdbc:  0.673 s
                withConnection:  0.646 s
              oldDbutilsMethods:  0.730 s
                 simpleMethods:  1.150 s
                 templateStyle:  4.566 s
               dataMapperStyle:  2.620 s
              activeRecordStyle:  2.828 s
      activeRecordDefaultContext:  2.768 s
```

可以看出与纯JDBC相比，各种Sql写法性能损失不大，最慢的是模板方法。

实体CURD方法与其它ORM工具的性能对比测试

BeetlSQL的作者闲大赋做了一个持久层工具性能对比测试项目, 见Dao-Benchmark项目或我的fork,偏重于测试实体的CRUD性能，不包括更底层的JDBC方法的性能测试。可以看到jSqlBox性能是领先的。

另外说一下，DAO工具的性能并不是太重要，也不用太拘泥于性能比较，因为瓶颈在数据库，而不是在DAO工具。DAO工具比拼的主要是易用性，引入DAO工具的目的是为了提高开发效率，而不是为了提高运行速度。

附录2：DAO工具对比

下表是jSqlBox在各个方面与一些其它DAO工具的对比，内容仅为个人看法，这些工具都在随时更新，而这篇文档却不一定随时更新，可能有说错的地方。表中的MP为MyBatis-Plus的缩写，此对比表内容较多，请以全屏模式打开。

对比项目	MP	BeetlSql	JFinal	jSqlBox	打分理由(仅参考官网文档，纯属个人看法)
总体架构	3.5	4	3.5	5	jSqlBox模块式架构更合理，各个模块可以抽取出来使用。MP内核基于MyBatis，这个内核比较大，本身固有的问题(如XML配置复杂)不能通过插件来解决。JFinal：基于Map的实体类不规范，不是标准POJO，对项目有侵入性
易学、易用、易维护	4	4	5	5	jSqlBox的配置和代码量在所有ORM工具中最少，见DaoBenchMark项目对比(缺JFinal)
从实体类生成DDL脚本	0	0	0	5	jSqlBox支持从实体创建跨数据库(80多种方言)的DDL脚本，更方便原型开发和单元测试
从数据库生成实体源码	5	5	5	5	代码生成器在项目后期作用有限，例如当实体类字段名重构，还是必须在IDE里打开源码手工重构，代码生成器帮不上忙。
分页	4	4	4	5	jSqlBox的分页无侵入性，任意SQL和CRUD方法都可以加入分页拦截器
事务	4	4	4.5	5	都兼容Spring事务，Jfinal和jSqlBox都自带声明式事务，可以甩掉笨重的Spring。jSqlBox的声明式事务有独立发布版，可以提供给其它工具使用。jSqlBox自带支持分库分表的分布式事务。
动态SQL	4	4	4	5	jSqlBox首创参数内嵌(Inline)式写法，方便拼接SQL，任意SQL和CRUD方法中都可以将SQL文本、参数、分页拦截器、缓存选项、分库分表选项等当作参数传递。其它DAO工具则主要依赖模板实现动态SQL，多绕了一层。
DBA友好(多行SQL存放)	3.5	4.5	5	4	BeetlSql不能切换模板。MyBatis的XML繁琐。jSqlBox缺省模板无语法功能，但它利用Java存放多行SQL，支持IDE定位。
支持重构的SQL	4	4	4	4.5	jSqlBox利用Q类或字符常量可以写出支持重构的SQL(Java8版)，而且没有重新发明新的SQL语法。BeetlSql中的Lambda查询器和MP中的条件构造器支持重构，但个人认为是重新发明SQL，意义不大。jFinal没有发明反模式，但也没有什么创新，所以打分不高。jSqlBox没有打满分是因为受Java语言限制，Lambda不能直接当参数传递，使用起来繁琐。
ORM关联查询	5	3	3	5	BeetlSql和JFinal实体关联查询有缺陷，例如dept=user.get("department")这种写法不支持重构，也没有自动装配POJO中的字段属性。JFinal中Blog.dao.find("select * from blog where user_id=?"，get("id"));这种方式隐含1+N问题
DataMapper模式	5	5	0	5	即类似dao.save(user)写法。JFinal不支持标准POJO的持久化。
ActiveRecord模式	5	0	5	5	ActiveRecord实现起来很简单，它是DataMapper模式的镜像，在jSqlBox中只有不到300行代码，但是BeetlSql没有集成这个功能，希望改进。jSqlBox的Java8版只需要声明ActiveEntity接口，不占用单继承。
接口代理模式	5	5	0	0	只需要写出接口，SQL用注释加在接口方法上，在运行时生成代理实例方法，因为考虑到这个功能比较复杂，而且MyBatis已经完备,而且作者本人并不喜欢这种模式，jSqlBox没有实现这个功能。但打零分并不表示jSqlBox不能使用这个模式，参见MyFat项目。

混合模型	?	5	5	5	即同时支持利用POJO和数据库字段存取，如user.setName(“Tom“).putTail(“usr_age“,10).save()。BeetlSql利用@Tail或TailBean实现，jSqlBox中的ActiveRecord支持混合模型，jFinal中的Model支持混合模型。MP不清楚，打个问号。
Record/Map模式	?	5	5	5	实体类不存在时，JFinal可以用Record类来直接存取数据库，jSqlBox用Tail类实现(是ActiveRecord的子类，源码只有两行)，BeetlSql还支持直接用Map来存取数据库。Tail/Record/Map这种方式不支持重构，所以不建议在项目中大量使用。
主从支持	2	5	2	5	jSqlBox的主从选择可以精细控制到任意一个CURD或SQL，jFinal需要手工选择主从库，不具备自动切换主从功能。MyBatis本身不支持主从，要利用AOP等第三方手段实现。
分库分表	2	2	2	4.5	只要使用了Sharding，对SQL写法就有侵入。jSqlBox自带基本的Sharding支持，支持原生SQL和所有数据库。其它DAO工具本身不具备sharding功能，需要第三方Sharding工具支持，限制较多。
批处理	4	4	4	5	除了通常的JDBC批处理方法外，jSqlBox还提供批处理开关模式。
缓存	5	5	5	5	都有，都是针对SQL查询结果而不是实体容器缓存
拦截器	5	5	5	5	都有
Sql日志	5	5	5	5	都有
乐观锁	5	5	5	5	都有
列名到实体字段映射	5	5	5	5	数据表与Java字段命名不符时，jSqlBox用@Column注解来更正，符合JPA惯例。jFina推荐Java字段命名与数据表一致。MP和BeetlSql有几种映射策略选择。
性能	4	5	?	5	对于持久层，易用性为王，不用多考虑性能。性能测试详见闲大赋的DaoBenchMark
文档	4	4	4	4	都有
知名度、成熟度	4	4	5	1	用的人越多，Bug就越少。成熟度反映了软件本身的质量，但它与软件的架构、功能、易用性无关。

上表中未加入Hibernate和MyBatis，是因为Hiberante太复杂、MyBatis不提供CRUD，缺点太明显。已经被很多人用脚投票了。
不加入EBean的原因是它的功能相对单调一些，设计思路与Hibernate相似，复杂性依然存在。
国产的NutzDAO功能也挺全的，而且也支持DDL脚本的生成，但因时间和篇幅不够，没有加入作对比。 JdbcTemplate和DbUtils等DAO工具因为功能单薄，也不加入作对比。
因时间有限，其它DAO工具的对比项仅参考官网文档,没有实测过，如有说错的也请指正。

附录3：版本发布记录

2017-12-04 1.0.0版发布

发布包依赖关系:jSqlBox1.0.0依赖于jdbpro1.7.0.1 + jdialects1.0.6 + jtransactions1.0.0(Optional), 其中jdbpro1.7.0.1依赖于commons-dbutils1.7

2018-03 1.0.7版发布

1.发布包依赖关系:jSqlBox1.0.7依赖于commons-dbutils1.7 + jdialects1.0.7 + jtransactions1.0.1 + jdbpro1.7.0.2(源码内嵌）
2.添加了Wiki说明，demo演示项目、SqlMapper风格以及多行文本支持

2018-06-21 2.0.0版发布

1.发布包依赖关系变化: 使用jSqlBox发布包只依赖于commons-dbutils1.7，其它子模块jDialects2.0.0、jTransactions2.0.0、jDbPro2.0.0以源码内嵌形式包含。
2.添加了分库分表、主从支持功能
3.demo目录下添加了jsqlbox-xa-atomikos分布式事务演示项目
4.demo目录下添加了jsqlbox-java8演示项目
5.将拦截器与ResultSetHandler分离，拦截器命名为SqlHandler
6.新增pXxxx系列方法，删除不实用的xXxxx系列方法，iXxxx系列方法改为不用ThreadLocal,旧的利用到ThreadLocal的SQL方法命名为INLINE大写方法
7.nBatch, nInsertBatch的<List>参数改成List<Object[]>

2018-07-27 2.0.1版发布

1. 整理并新增了许多CRUD方法, 增加了按Sample查询。
2. 完善了实体ORM查询，允许多步查询、添加findRelated系列方法、添加树结构查询演示。
3. JSQLBOX中的静态方法去除g开头字母。
4. 完成了整个项目的开发。将转入Bug发现清除阶段。

2018-07-30 2.0.2版发布

只进行了性能调优，将实体关联查询的速度提高了大约10%

2018-08-21 2.0.3版发布

因为jBeanBox内核重写，引入了@AOP注解，相应地jSqlBox的事务配置方式有变化。

2018-11-18, 2.0.4版发布

1. 添加了从数据库生成实体源码功能（在jDialects模块）
2. 添加了tail功能(在jSqlBox模块的ActiveRecord类中)，这个功能对应BeetlSql的Tail或jFinal的Record功能，新增了putTail和getTail方法。原来ActiveRecord中的put和putFiels方法取消，取而代之的是putFields和forFields方法
3. 添加了对三个常用MVC框架ActFramework、SpringBoot、jFinal的整合示例,以及与MyBatis混用的示例。

2018-12-17, 2.0.5版发布

1. 添加了对三个JPA注解@Version, @Enumerated, @Convert的支持
2. 事务模块(jTransactions)添加了手工事务ManualTx类，并且在TinyTx类中添加了beginTransaction、commit、rollback三个方法。
3. MyFat1.0.0版插件发布，它将jSqlBox2.0.5作为插件集成到MyBatis3.4.6版中去。

2019-1-31, 2.0.6版发布

1. 事务模块(jTransactions)添加了GroupTx类，这是一个多数据源事务工具，它与分步式事务不同，仅限于一组数据源中只能有其中一个（具体不知哪一个)提交的场合，最适用的场景为Sharding分库的事务。

2019-9-04, 3.0.0版发布

1. 事务模块做了较大的修改，新增了GtxConnectionManager类，用于支持分布式事务。
2. demo下的BeetlSql示例改成直接使用Beetl3.0模板
3. JAVA8方法引入$,a$,c$简化写法
4. JDBPRO中新增noNull方法。
5. 一些其它改进等，如TinyTx被注记不再使用，改为TinyTxAOP

2020.2.11, 4.0.0版发布

1.从4.0.0起，把所有依赖都打包进来，jSqlBox只有一个单独的jar包了，不再需要依赖任何第三方工具。分为jre8和jre6两个包发行。
2.数据库方言从原来的枚举类型改为普通类的形式，以便于用户自已添加新的数据库方言。
3.实体映射新增了Calendar日期类，并新增了7种Java8日期类型支持。
4.更正了在Oracle数据库下java.util.Date日期映射出错的问题，并增加了对JPA标准注解@Temporal的支持。

2020.2.11, 4.0.1版发布

改正了@Column标签与JPA定义不符的问题，在columnDefinition里可以包含额外的DDL片段。

2020.2.23, 4.0.2版发布

增加了@CreateTimestamp、@UpdateTimestamp、@CreatedBy、@LastModifiedBy、@COLUMN五个注解支持。

2020.3.1, 4.0.3版发布

1.更正setAuditorGetter方法为绑定一个实例，而不是利用jBeanBox绑定一个类，以达到精细控制每一个DbCotnext都可以设定自已的AuditorGetter，而不是共用一个。
2.增强了从数据库自动生成实体源码功能，加入了linkedStyle和fieldFlags两个选项。
3.增加了一个SqlOption.IGNORE_EMPTY功能，如果实体操作加入这个参数，则实体插入和更新时会忽略值为null或为空字符串的字段。

2020.4.13, 4.0.6版发布 有以下内容变更：

1. 从数据库生成Java源码时，char(xx) 类型不再映射成单个Character字符类型, 而是映射成String类型.
2. DB类中增加一个静态iPrepare方法以方便使用。
3. 新增一个SQL工具类，保存了SQL关键字常量，以减少拼写SQL的错误和去除字符引号，如字符" select " 可以用SQL.SELECT代替，例如以下SQL:
   User u=DB.eLoadBySQL(User.class, "select * from user_tb where user_id=", ques(3), " or user_name=", ques("other"));
   如果配合源码自动生成的列名常量，以及静态引入的SQL常量，则可以写成以下支持SQL重构的形式：
   User u=DB.eLoadBySQL(User.class, SELECT_STAR, FROM, User.TABLE_NAME, WHERE, User.ID, EQ, ques(3), OR, User.NAME, EQ, ques("other"));
   这种写法相比与JOOQ之类的SQL工具来说，优点是秒懂、学习负担轻，因为本质上jSqlBox就是在Java里拼接SQL字符串。
4. DbContext和DB类中新增了一个eFindOneBySQL方法，这个方法如果没有发现实体将会返回null, 而不是象eLoadBySQL方法一样抛出异常。
5. StrUtils工具类中增加了一个array方法，用于将数组转换写成("abc", 12, 345)类型的字符串形式。
6. TableModelUtilsOfDb和TableModelUtilsOfJavaSrc中，改成用字符串常量的形式，并有功能调整，
7. 允许使用@Column(name=" ` xxx ` ") 的方式使用数据库关键字作为列名，支持反单引号、双引号、中括号三种引用符，视不同的数据库而定。使用数据库关键字作为列名这种做法不推荐，通常只用于旧项目改造，新项目强烈不建议使用关键字作为列名。
8. TypeUtils中更正了实体不支持基本数据类型如int、short等类型的bug
9. 更正了@IdentityId注解在MySql下出错的bug，这是BigInteger类型转换有bug造成的
10. 更正了@Enumerated注解的字段应该允许为null

2020.6.25, 4.0.7版发布 有以下内容变更：

1.增加tx和tryTx两个事务模板方法
2.增加缓存翻译功能

2020.7.26, 4.0.8版发布 有以下内容变更：

增加导出数据库结构为Excel格式,增加导出实体类结构为Excel格式，详见jDialects子项目中的使用说明第5.4节“实体或数据库结构导出Excel”

2020.10.15 5.0.1.jre8版发布，有以下内容变更:

- 去除pinte方法, i系统方法改为qry/ins/exe/upd系列方法, e系列方法改为entity打头,t系列方法改为传入模板，p和n这两种写法因为很少用，所以直接取消。
- 去除DB.sql()方法，默认字符串都是SQl片段
- 大写的PARAM、QUES、VALUESQUESTION方法去掉
- PrintSqlhandler拦载器改进可以输出SQL参数代入后的完整SQL，方便粘贴到调试工具里运行
- 添加DB.other方法
- 添加@uuid26注解
- 类型转换做成可配置
- StrUtils工具类中的array静态方法，当条目为空时返为(null)
- 新增DB.qryList方法，返回查询内容的第一行内容
- 新增DB.qryMap方法，返回查询内容的第一列内容
- 添加when方法

- 去除JDBPRO类，只保留DB

2020.11.21 5.0.3.jre8版发布:

- 添加setTenantGetter方法，实现动态多租户功能
- 更正TableModelUtilsOfDB类中没有加catalog导致获取的TableModel列多出来的问题
- TableModelUtilsOfDB添加了compareDB方法，用于比较两个数据库结构是否有差异

2021.01.13 5.0.4.jre8版发布:

1.如上所述，新增生成和使用Q类来写出支持重构的SQL功能。 2.DbContext新增一个SqlItemhandler配置选项，可以用来定制自己的Sql条目解析器。 3. DbPro类中新增qryBooleanValue方法 4.DB类中新增notBlank和noBlank两个方法，用来在动态拼接SQL时判断空字符串条件，和原有的notNull、noNull使用方法类似

2021-07-02, 5.0.5.jre8版发布

在ImprovedQueryRunner类（是DbContext的基类)中添加了StatementConfiguration初始化参数，以保持与DbUtils的初始化参数一致。

2021-07-17, 5.0.6.jre8版发布

改正jDialects中生成实体源码时getter和setter中的bug, 新增一个CamelHandler，用于将下划线格式的结果转化为驼峰式结构

2021-07-17, 5.0.9.jre8版发布

在DialectFunctionTemplate中添加registerFunction方法，用来登录自定义的函数，并添加一个通用qt函数，用于在不同数据库下使用时添加对应数据库的引用符号。

2022-02-27, 5.0.10.jre8版发布

1.更新junit和HikariCP版本号。 2.更正IdentityGenerator在PostgreSQL中的DDL生成bug。 3.在jDialect中添加全局命名规则NamingConvension类，并设缺省值为NamingConvension.NONE 4.添加一个@UUID注解，以方便使用，这个注解和@UUID32一样，都是32位字符长的随机ID

2022-02-27, 5.0.11.jre8

1.修复Decimal类型bug 2.添加tableTail 3.jDialects中修复Mysql中不支持TimeStamp的bug

2022-02-27, 5.0.12.jre8

jDialects模块增加运行期动态增、删列的功能

2022-04-, 5.0.13

1.合并jDialects中Anchor.lu的提交pr#3，关于数据类型的添加。
2.改进getNextId方法签名，IdGenerator接口改变，不再需要依赖NormalJdbcTool接口 3.添加BeeCP数据库连接池的测试

FAQ 常见问题

自定义实体-数据库表名/列名的命名转换规则

这个在5.0.10.jre8版之后，可以在程序启动时，使用全局静态方法Dialect.setGlobalNamingConvension()方法来设定，示例如下：

```
//缺省设定，不作改变，如 OrderDetail.class映射到OrderDetail数据库表， orderPrice映射到orderPrice数据列
Dialect.setGlobalNamingConvension(NamingConvension.NONE);

//对应数据库的小写下划线形式，如 OrderDetail.class映射到order_detail数据库表， orderPrice映射到order_price数据列
Dialect.setGlobalNamingConvension(NamingConvension.LOWER_CASE_UNDERSCORE);

//对应数据库的大写下划线形式，如 OrderDetail.class映射到ORDER_DETAIL数据库表， orderPrice映射到ORDER_PRICE数据列
Dialect.setGlobalNamingConvension(NamingConvension.LOWER_CASE_UNDERSCORE); 变
```

如果以上三种形式不能满足要求，可以参考NamingConvension源码，写一个自定义的NamingConvension对象类并在setGlobalNamingConvension中设定它的实例即可。

另外说明一下，使用了setGlobalNamingConvension方法之后，对于个别的类和实体属性，依然可以用@Table和@Column这两个注解来调整命名，注解有更高的优先级。

实体类java.util.Date保存时分秒丢失

java.util.Date是java.sql.Date，java.sql.Time和java.sql.Timestamp的父类，jSqlBox和Hibernate一样，默认映射为java.sql.Date，如要保存时分秒可采用@Temporal注解来注明，这是JPA的一个标准注解，jSqlBox也自带了:

```
@Temporal(TemporalType.TIMESTAMP)
java.util.Date d5;
```

详见com.github.drinkjava2.jsqlbox.function.jdialects.typemapping.DateTimeTest中的testD5和testD12方法示例;

另外，关于日期转换还可以用dbContext.setJavaToJdbcConverter()/setJdbcToJavaConverter()方法来进行全局的配置，这是不太常用，它是针对底层SQL数据转换的一种全局配置方案。

另外还可以有@Convert方案和采用Java8日期类型方案，但都不如@Temporal(TemporalType.TIMESTAMP)注解来得方便。

在SQL中根据不同的方言对关键字添加引用符号如双引号、反单引号等

这个在5.0.9.jre8版之后，可以用qt函数解决，如：

```
String sql = DB.trans("select qt(name) from qt(helloWorld) where qt(name) like ?");
String result=DB.qryString(sql, DB.par("He%"));
```

即可针对不同的数据库，在对应的字段上添加与该数据库对应的引用符号。

未命名的页面(1)

.keep