Fire 数据库操作工具

使用说明

（JSPGen4.0 工具包）

联系方式

QQ 群：122818143

在线 API 文档： http://help.jspgen.com/api/fire4/

www.JSPGen.com

二零一四年三月

(共 36 页)

目 录

一、介绍

Fire（火）：是一个用于 Java 编程的数据库操作工具，轻巧简单实用，含数据库连接对象、数据库访问对象及代码生成三部分；由 JSPGen 软件开发框架第 4.0 版时提出（前身为 JSPGen3.0 时的数据库连接管理工具），是对传统 JDBC 数据库操作进行的二次封装，支持数据缓存处理、支持在同一项目中对多数据库进行操作，更重要的是对一些连接池进行了集中配置，以此来简化对数据库的操作。

其中，SQL 操作部分支持变量定义，整个代码简洁直观，稍懂 SQL 就可以胜任，无过高学习成本。

注：单独运行时，需Grapes工具包支持（Grapes：包含多种常用工具，解决平时编程会经常遇到的问题，减少重复劳动，由JSPGen软件开发框架第 4.0 版时提出。）；

以下内容中【绿色】为我们需要留意的关键内容(不含代码注释)，【红色】为我们可修改的内容，【蓝色】为提醒色或为不可更改的内容(不含代码注释)，【灰色】为某种功能效果的另一种实现方式。

1.1、功能

1、支持主流数据库读取和写入操作（理论上支持 JDBC 所支持的所有数据库）；

2、支持多数据库配置、多连接池配置、数据库事务处理(含断点设定)、支持多个不同数据库同时操作；其中，连接池配置默认支持：BoneCP、Proxool 连接池（可通过接口扩展实现其它连接池）；

3、支持数据查询结果集自动转换成 List、Map 或 Entity（数据表实体类 Bean）对象；

4、支持 SQL 原始拼装、SQL 原始传参式赋值及 SQL 变量索引式赋值（变量类型：数字索引、字符索引、数字字符混合式索引及 Map 或 Entity 对象赋值）；其中，若采用原始传参式赋值或变量定义式赋值，可过滤不安全字符；

5、支持 SQL 语句运行详情记录至日志文件（含 SQL 语句及参数赋值情况），属可配置功能；

6、支持 Java 开发模型层常见设计：Entity+Dao+Service 结构，各接口的实现类可根据文件路径动态读取，基础文件（Entity、Dao、Service）可根据数据库各表结构由工具生成；

7、支持数据库各表命名前缀及间隔符变更操作（只需修改配置文件便可，无需更改操作文件）；

8、支持数据库各表主键值自动填充功能（支持：id 型，属自增式整型数字、uuid 型，属 32 位无重复字符串）。

1.2、约定

友情提醒：有看不明白的地方，可先跳过，放到最后再返回查看。

1、数据表设计

A、表名及字段名统一为小写字母，单词与单词间采用支持的特殊符间隔，如：_下划线；

B、数据表主键可由程序生成写入实体类(数据表)，不建议使用数据库自带主键填充功能，不建议使用数据库外键关联功能；

C、若实体类继承于 Fire 实体基类，则数据表中主键名称需与统一类型为：varchar(32)；

D、数据库编码统一为：UTF-8 编码。

2、Java 中 SQL 语句设计

A、表实体类属性书写：数据表字段定义为 **user_name**，则 Java 实体类对应字段属性应该定义为 **userName**；

B、SQL 字段书写：语句中字段名称均以数据表设计的字段名称为准，如：数据表字段定义为 **user_name**，则 SQL 语句调用时为（select **user_name** from 表名 where user_name = ?）；

C、SQL 变量赋值：若采用变量索引赋值，则赋值对象 Map 中的 key 元素或实体类字段属性需要与变量索引命名一致，可按驼峰命名法命名，如：赋值对象元素为 **userName**，则变量定义须为 **:userName** (select **user_name** from 表名 where user_name = **:userName**)。

D、数据表名格式化：数据表名有三个关键属性：1、表前缀；2、间隔符、3、名称，前两个均可在配置文件中配置，所以在使用 Fire 提供的默认 Dao 实现类指定数据表时可直接指定第 3 个属性便可（支持峰驼式命名，可自动格式化为数据表实际表名），详细说明参见 4.2 章节演示 Dao 实现类初始化时的注释说明。

1.3、Fire 下载

JSPGen 官网：http://www.jspgen.com/

在项目中使用需要加入以下文件（可直接在官网下载第三方 Jar 包）：

jspgen-grapes-4.0.jar、jspgen-fire-4.0.jar、mysql-connector-java-5.1.7-bin.jar

bonecp-0.7.1.jar 、dom4j-1.6.1.jar、commons-collections-3.2.jar

log4j-1.2.16.jar、slf4j-api-1.7.2.jar、slf4j-log4j12-1.6.1.jar

1.4、FireAPI

在线地址：http://help.jspgen.com/api/fire4/

二、配置

配置文件位于项目类文件跟目录，如下所示：

【必备】数据源配置文件：/WEB-INF/classes/ fire-config.xml

【可选】Proxool 连接池属性文件：/WEB-INF/classes/ proxool.properties

2.1、数据源配置（含连接池配置：BoneCP & Proxool）

数据源配置一共有两部分：

第一部分，Fire 属性配置（必备）；

第二部分，各连接池自身属性配置（可选，属性详情可参考各连接池自身说明）。

其中，在 Fire 属性配置中，数据源名称在整个文件中要保持唯一性（默认：JSPGen）；

关于连接池配置，默认支持 BoneCP 连接池与 Proxool 连接池，我们推荐采用 BoneCP 连接池。

A、BoneCP 配置方式（推荐）：

```
<?xml version="1.0" encoding="UTF-8"?>
<fire version="4.0">
  <!-- 数据源名称（默认：JSPGen）、是否启用 -->
  <props name="JSPGen" status="true">
    <!-- Fire属性配置 开始 -->
    <!-- 数据库间隔符(表名、字段与实体类转换需要) -->
    <prop name="fire.connection.separator">_</prop>
    <!-- 数据库表前缀 -->
    <prop name="fire.connection.table.prefix"></prop>
    <!-- 连接池实现类 -->
    <prop
name="fire.connection.provider">fire.connection.provider.BoneCPConnectionP
rovider</prop>
    <!-- 数据库Driver -->
    <prop name="fire.connection.driver">com.mysql.jdbc.Driver</prop>
    <!-- 数据库URL -->
    <prop
name="fire.connection.url"><![CDATA[jdbc:mysql://127.0.0.1:3308/test?useUnic
ode=true&characterEncoding=UTF-8]]></prop>
    <!-- 数据库用户名 -->
    <prop name="fire.connection.username">root</prop>
    <!-- 数据库密码 -->
    <prop name="fire.connection.password">root</prop>
    <!-- 每条SQL语句都被记录到日志 -->
    <prop name="fire.connection.logsql">true</prop>
    <!-- Fire属性配置 结束 -->
    <!-- BoneCP连接池属性配置 开始 -->
    <!-- 连接池中未使用的链接最大存活时间，单位是分，默认值：60，如果要永远存活设置为0 -->
    <prop name="bonecp.idleMaxAgeInMinutes">1</prop>
    <!-- 检查连接池中空闲连接的间隔时间，单位是分，默认值：240，如果要取消则设置为0 -->
    <prop name="bonecp.idleConnectionTestPeriodInMinutes">1</prop>
    <!-- 分区数 ，默认值2，最小1，推荐3-4，视应用而定-->
    <prop name="bonecp.partitionCount">1</prop>
    <!-- 每个分区最小的连接数 -->
```

```
    <prop name="bonecp.minConnectionsPerPartition">1</prop>
    <!-- 每个分区最大的连接数 -->
    <prop name="bonecp.maxConnectionsPerPartition">5</prop>
    <!-- 每个分区中连接增长数量，默认为1 -->
    <prop name="bonecp.acquireIncrement">1</prop>
    <!-- 每次去拿连接的时候一次性要拿几个,默认值：2 -->
    <prop name="bonecp.acquireIncrement">1</prop>
    <!-- 设置statement缓存个数。这个参数默认为0 -->
    <prop name="bonecp.statementsCacheSize">50</prop>
    <!-- 每个分区释放链接助理进程的数量，默认值：3，除非你的一个数据库连接的时间内做了很多工作，不然过多的助理进程会影响你的性能 -->
    <prop name="bonecp.releaseHelperThreads">3</prop>
    <!-- BoneCP连接池属性配置 结束 -->
  </props>
  <!--其它数据库连接、连接池配置 -->
  <!-- … -->
</fire>
```

B、Proxool 配置方式：

Proxool 连接池由于采用驱动方式实现管理，所以数据库连接地址可在 Fire 中配置，也可在 Proxool 属性文件中配置（此处优先级高于 Fire 中配置），最后在 Fire 配置中调用 Proxool 配置前缀即可，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<fire version="4.0">
  <!-- 第一种配置方式 -->
  <!-- 数据源名称（默认：JSPGen）、是否启用 -->
  <props name="JSPGen1" status="true">
    <!-- Fire属性配置 开始 -->
    <!-- 数据库间隔符(表名、字段与实体类转换需要) -->
    <prop name="fire.connection.separator"></prop>
    <!-- 数据库表前缀 -->
    <prop name="fire.connection.table.prefix"></prop>

    <!-- 连接池实现类 -->
    <prop
name="fire.connection.provider">fire.connection.provider.ProxoolConnection
Provider</prop>
    <!-- 数据库Driver -->
    <prop name="fire.connection.driver">com.mysql.jdbc.Driver</prop>
    <!-- 数据库URL -->
    <prop
```

```
name="fire.connection.url"><![CDATA[jdbc:mysql://127.0.0.1:3308/test?autoReconnect=true&useUnicode=true&characterEncoding=UTF-8]]></prop>
    <!-- 数据库用户名 -->
    <prop name="fire.connection.username">root</prop>
    <!-- 数据库密码 -->
    <prop name="fire.connection.password">root</prop>
    <!-- 每条SQL语句都被记录到日志 -->
    <prop name="fire.connection.logsql">false</prop>
    <!-- Fire属性配置 结束 -->
    <!-- Proxool连接池属性配置 开始 -->
    <!-- 指定Proxool属性文件 -->
    <prop name="proxool.file">proxool.properties</prop>
    <!-- 指定Proxool属性文件中的前缀符 -->
    <prop name="proxool.prefix">jdbc-0</prop>
    <!-- Proxool连接池属性配置 结束 -->
  </props>

  <!-- 第二种配置方式 -->
  <props name="JSPGen2" status="false">
    <!-- Fire属性配置 开始 -->
    <!-- 数据库表前缀 -->
    <prop name="fire.connection.table.prefix"></prop>

    <!-- 数据库实现类 -->
    <prop
name="fire.connection.provider">fire.connection.provider.ProxoolConnectionProvider</prop>
    <!-- 每条SQL语句都被记录到日志 -->
    <prop name="fire.connection.logsql">false</prop>
    <!-- Fire属性配置 结束 -->
    <!-- Proxool连接池属性配置 开始 -->
    <!-- 指定Proxool属性文件 -->
    <prop name="proxool.file">proxool.properties</prop>
    <!-- 指定Proxool属性文件中的前缀符 -->
    <prop name="proxool.prefix">jdbc-1</prop>
    <!-- Proxool连接池属性配置 结束 -->
  </props>
  <!--其它数据库连接、连接池配置 -->
  <!-- … -->
</fire>
```

C、Proxool 属性文件：

```
#连接池别名(在整个Tomcat环境中保持唯一)
jdbc-0.proxool.alias=jspgen1
```

```
#以下属性可在Fire配置属性中配置
#jdbc-0.proxool.driver-class=com.mysql.jdbc.Driver
#jdbc-0.proxool.driver-url=jdbc:mysql://127.0.0.1:3308/test?autoReconnect=true&useUnicode=true&characterEncoding=UTF-8
#jdbc-0.user=root
#jdbc-0.password=root
#自动侦察各个连接状态的时间间隔(1秒=1000毫秒)，侦察到空闲的连接就马上回收，超时的销毁(当创建新连接时才开始操作)
jdbc-0.proxool.house-keeping-sleep-time=60000
#最大的等待请求数，未有空闲连接可以分配而在队列中等候的最大请求数，超过这个请求数的用户连接不会被接受
jdbc-0.proxool.simultaneous-build-throttle=10
#一次产生连接的数量(正式库10，测试库1)，不能超过最大连接数
jdbc-0.proxool.prototype-count=1
#最大连接数(正式库700，测试库10)，默认15个，超过这个数，再有请求时，就排在队列中等候，等待最大请求数由maximum-new-connections决定
jdbc-0.proxool.maximum-connection-count=10
#最小连接数(正式库350，测试库1)，默认5个
jdbc-0.proxool.minimum-connection-count=1
#连接最大活动时间，默认5分钟(300000)，单位毫秒
jdbc-0.proxool.maximum-active-time=60000
jdbc-0.proxool.verbose=true
jdbc-0.proxool.trace=false

#连接池别名(在整个Tomcat环境中保持唯一)
#jdbc-1.proxool.alias=jspgen2
# … 与上类似，只是前缀不同
```

2.2、接口实现类调用

主要为 Java 开发模型层服务（Entity+Dao+Service），各接口实现类可根据文件路径动态读取。

A、默认读取方法：

```
/**
 * 获取Dao实现类(dao)
 * @param name 路径前缀(默认：jspgen)
 * @param filename 类文件名(实际文件中开头字母应大写)
 */
getDaoImpl(String name, String filename);
getDaoImpl(String filename);

/**
 * 获取Service实现类(service)
```

```
 * @param name 路径前缀(默认：jspgen)
 * @param filename 类文件名(实际文件中开头字母应大写)
 */
getServiceImpl(String name, String filename);
getServiceImpl(String filename);
```

B、Java 调用方式： (* 为实际文件命名前缀，使用时需替换为实际文件名)

```
private *Dao *Dao = (*Dao) FireImpl.getImpl(“dao”,"*DaoImpl"); 或
private *Dao *Dao = (*Dao) FireImpl.getDaoImpl("*DaoImpl");
private *Service *Service = (*Service)
FireImpl.getImpl(“service”,"*ServiceImpl"); 或
private *Service *Service = (*Service) FireImpl.getServiceImpl("*ServiceImpl");
```

三、数据写入、数据读取及事务处理

调用方法，可参考第 6.1 章节 API，操作前需注意如下事项：

A、变量索引：格式为：**字段名**(峰驼式命名)，采用变量索引赋值时，变量索引左右不能有单引号、双引号；

B、批量赋值：批量赋值元素优先级高于单个赋值元素，若采用数字索引，则索引从 0 开始；

C、索引支持：数字索引、字符数字索引、字符变量索引、混合索引；

D、操作流程： (开启事务) → **创建赋值**(创建 SQL 语句 → 参数赋值) → **查询或更新** → (提交事务)。

3.1、数据写入

先确定当前操作表名：

组装当前数据表名：String tableName = FireAccess.CONFIG_PREFIX + **test_table**;

组装后，得 **:prefixtest_table**，表前缀占位符 + 数据表名，也可按此规则直接书写。

FireAccess fa = FireBuild.getAccess("**JSPGen**"); // 获取某数据源访问对象（JSPGen为所配置的数据源名称，也是默认数据源名称）

// Connection conn = fa.getConnection()；// 获取当前数据库连接，方便直接操作的朋友使用

A、原始拼接

String sql = "insert into "+tableName+" (title ,hits) values ('JSPGen 软件开发框架','21')";

System.out.println("Update1："+**fa.executeUpdate(sql)**); // 更新操作

操作流程：一步完成（执行语句不安全）。

B、单个赋值（推荐）

String sql = "insert into "+tableName+" (title, hits) values (?, ?)";

fa.createSQL(sql);　　// a：创建 SQL 语句

fa.setParameter(**0**, "JSPGen软件开发框架").setParameter(**1**, "21"); // b：赋值操作（数字索引，从 0 开始）

// String sql = "insert into "+tableName+" (title, hits) values (**:title**, **:hits**)"; // 变量无单双引号

// 数字索引、字符数字索引

// fa.createSQL(sql).setParameter(**0**, "JSPGen 软件开发框架").setParameter(**“1”**，"21");

//字符变量索引

// fa.createSQL(sql).setParameter(“title”，"JSPGen 软件开发框架").setParameter(“hits”，"21");

// fa.setParameter("hot", "13");　　// 即便有多余的也无妨

// 混合式索引

// fa.createSQL(sql).setParameter(**0**，"JSPGen 软件开发框架").setParameter(**“hits”**，"21");

System.out.println("Update2: "+**fa.executeUpdate()**); // c：更新操作

操作流程：先创建SQL语句，再赋值，最后更新（执行语句安全）。

C、批量赋值(Map 集合)（推荐）

Map<String, Object> prame = new HashMap<String, Object>();

String sql = "insert into "+tableName+" (title, hits) values (?, ?)";

prame.put("0", "JSPGen 软件开发框架"); prame.put("1", "21");

fa.setSQLParameter(**sql, prame**); // a：批量赋值操作

// String sql = "insert into "+tableName+" (title, hits) values (:title, :hits)";

// prame.put("title", "JSPGen 软件开发框架"); prame.put("hits", "21");

// fa. createSQL(**sql**).setParameter(**prame**);

System.out.println("Update3: "+**fa.executeUpdate()**); // b：更新操作

操作流程：创建SQL语句与赋值一起处理，再更新（执行语句安全）。

D、批量赋值(Entity 实体类集合）（推荐）

String sql = "insert into "+tableName+" (title, hits) values (:title, :hits)";

*Entity prame = new *Entity (); // 实体类 Bean，实际使用中，*替换为实际文件名称

prame.setTitle("JSPGen 软件开发框架"); prame.setHits(21);

fa.setSQLParameter(**sql, prame**); // a：批量赋值操作

// fa. createSQL(**sql**).setParameter(**prame**);

System.out.println("Update4: "+**fa.executeUpdate()**); // b：更新操作

操作流程：创建SQL语句与赋值一起处理，再更新（执行语句安全）。

注：更新操作、删除操作与写入操作类似，仅仅 SQL 语句稍有不同：

sql = "insert into "+tableName+" (title, hits) values (**:title**, **:hits**)"; // 插入语句

sql = " update "+tableName+" set title=**:title**, hits= **:hits** where id=**:id**"; // 更新语句

sql = " delete from "+tableName+" where id=**:id**"; // 删除语句

3.2、数据读取

组装当前数据表名：String tableName = FireAccess.CONFIG_PREFIX + **test_table**;

FireAccess fa = FireBuild.getAccess("**JSPGen**");

A、返回原始结果集

```
String sql = " select id,title,hits from "+tableName+" where hits = 21";
ResultSet rs = fa.executeQuery(sql); // 运行查询
// String sql = " select id,title,hits from "+tableName+" where hits = :hits";
// fa. fa.createSQL(sql).setParameter( “hits”, 21);
// ResultSet rs = fa.executeQuery(); // 运行查询
System.out.println("获取结果集总数并不关闭结果集："+fa.last(rs, false));
// 第一个参数为空，则采用自身最近一次查询结果集；第二个参数为是否关闭结果集，默认是关闭。
```

B、返回 List & Map（结果集封装）

```
fa.createSQL("select id,title as name,hits as age from "+tableName+" where id<=100");
List<Map<String, Object>> list = fa.list();
// System.out.println("获取结果集列表并关闭结果集："+fa.list(rs, true).toString());
// 第一个参数为空，则采用自身最近一次查询结果集；第二个参数为是否关闭结果集，默认是关闭。
for(int i=0; i<list.size(); i++){
    Map<String, Object> person = (Map<String, Object>)list.get(i);
    if(person != null) {
        System.out.println("用户名："+person.get("name")+" 年龄："+person.get("age"));
    }
}
```

C、返回 List & Entity（结果集封装）

```
fa.createSQL("select id,title as name,hits as age from "+tableName+" where id<=100");
List<Person> list = fa.list(Person.class); // 对应实体类的Bean对象
for(int i=0; i<list.size(); i++){
    Person person = list.get(i);
    if(person != null) {
        System.out.println("用户名："+person.getName()+" 年龄："+person.getAge());
    }
}
```

D、返回单条信息

```
fa.createSQL("select id,title as name,hits as age from "+tableName+" where hits=21");
Map<String, Object> person = fa.unlist()；// 获取单条信息
System.out.println("用户名："+person.get("name")+" 年龄："+person.get("age"));
// Person> person = fa.unlist(Person.class); // 对应实体类的 Bean 对象
// System.out.println("用户名："+person.getName()+" 年龄："+person.getAge());
```

E、返回结果集中某列数据集合

```
fa.createSQL("select id,title as name,hits as age from "+tableName+" where id<=100");
List<Object> list = fa.result(1); // 结果集序列，从 1 开始，默认为 1
// List<Object> list = fa.result(1，rs，false);
// 第一个参数为获取的序列，第二个为结果集对象，第三个为处理后是否关闭结果集
// 类似方法：Object obj = fa.unresult(2); // 获取第二列的单条数据
for(int i=0; i<list.size(); i++){
```

```
        Object obj = list.get(i);
        if(obj != null){
            System.out.println("NotEmpty"); }else{ System.out.println("isEmpty");
         }
}
```

3.3、事务处理

组装当前数据表名：String tableName = FireAccess.CONFIG_PREFIX + **test_table**;

```
FireAccess fa = FireBuild.getAccess("JSPGen");
try{
    fa.begin();  // 开启事务
    System.out.println("事务 start: "+fa.isBegin());
    Map< String, Object> param = new HashMap< String,  Object> ();
    param.put("id", 100);  param.put("title", Dates.getTimeMillis());
    fa.createSQL("insert into "+tableName+" (id, title) values (:id, :title)");
    fa.setParameter(param);  System.out.println("事务 1: "+fa.executeUpdate());
    // 数据库更新操作，返回影响行数(不受后面出错语句影响，但数据库数据会回滚还原)
    // 错误执行
    param = new HashMap< String,  Object> ();
    param.put("id", 200);  param.put("name", Dates.getTimeMillis());
    fa.createSQL("insert into "+tableName+"x (id, title) values (:id, :title)");
    fa.setParameter(param);  System.out.println("事务 2: "+fa.executeUpdate());
```

```
        // 数据库更新操作，在这里故意写错表名，运行完程序，看操作 1 成功没
        fa.commit(); //事务提交
        System.out.println("1：commit");
        System.out.println("事务 end: "+fa.isBegin());
        System.out.println("listSQL："+fa.getSQLParameters().toString());
        // 当前运行的所有 SQL 列表（含参数集合），当前数据库连接关闭前有效。
} catch (Exception e) {
        fa.rollback(); // 操作不成功则回滚
        System.out.println("2：rollback");
        // e.printStackTrace();
} finally {
        System.out.println("3："+fa.isClose());
        fa.close(); // 关闭当前数据库连接
        System.out.println("4："+fa.isClose());
}
```

四、演示

演示均以添加数据、查询数据为例，更新操作、删除操作与添加操作类似，仅仅 SQL 语句稍有不同。

注：SQL 语句所有的字段均以数据库设计的字段命名出现，无组装情况，唯数据表名称有组装情况，具体情况参见 4.2 章节演示 Dao 实现类初始化时的注释说明。

4.1、JSP 中直接使用

```
<%@page pageEncoding=" UTF-8"%><%@page import="java.util.*"%>
<%@page import="fire.FireBuild"%><%@page import="fire.FireAccess"%>
<%
    String tableName = FireAccess.CONFIG_PREFIX + “test_table”;
    FireAccess fa = FireBuild.getAccess("JSPGen");
    try {
        Map<String, Object> param = new HashMap<String, Object>();
        param.put("id", 10);
        param.put("title", Dates.getTimeMillis());
        fa.createSQL("insert into " + tableName + " (id, title) values
(:id, :title)");
        fa.setParameter(param);
        System.out.println("Update: " + fa.executeUpdate());
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        fa.close(); // 关闭当前访问对象
    }
%>
```

4.2、Java 开发模型层使用（设计模式：Entity+Dao+Service）

在这个设计模式中： Entity 层为数据模型定义，Dao 层为数据原子操作(数据读取和数据写入)，直接和数据库打交道，Service 层为业务逻辑层（调用 Dao 方法，基类中含 Dao 层通用方法)。

通常业务不复杂的直接使用 Dao 层，业务逻辑比较复杂再加 Service 层。

在 Fire 所提供的 Dao 层中已帮我们实现了比较通用的 CRUD 操作方法【添加(Create)、查询(Retrieve)、更新(Update)和删除(Delete)】，我们可以直接使用；同时，我们也可以新建文件分别继承于 Fire 所提供的 Entity、Dao、Service 各基类接口文件，这样只需要在我们新建的文件实现类中增加自身需要的方法即可，无需再重写 CRUD 操作方法。

Fire 默认提供了两种类型的基类文件（CRUD 通用方法），一个是实体类型的，一个是 Map 集

合型的。

实体类型的对传递的参数有长度限制（长度由实体类字段属性数量决定），但后期维护方便。若需要修改实体类属性命名，使用 Eclipse 编程工具修改命名后，可自动将使用该实体类关联的属性一起修改；Map 集合型对传递参数无长度限制，如果业务逻辑比较复杂，后期维护会很麻烦。若需要修改某 key 健命名，凡采用这个集合的接收方都需要修改，如果仅仅是与数据表关联，那问题倒不大。

实际业务使用中大家可以根据情况选择继承不同的基类文件来获取 CRUD 通用方法（所有方法可参见第 6.2 章节 API）。

Fire 所提供的基类文件路径分别为（通常小项目中接口文件可省略）：

实体类型接口文件：fire.entity. Entity.java、fire.dao.Dao.java、fire.service. Service.java

Map 型接口文件：fire.dao.DaoMap.java、fire.service. ServiceMap.java

实体类型实现类： fire.dao.impl.DaoImpl.java、fire.service.impl. ServiceImpl.java

Map 型实现类：fire.dao.impl.DaoMapImpl.jav、fire.service.impl. ServiceMapImpl.java

下面我们以实体类方式进行演示，新建基础文件（为节省篇幅，无接口文件，均继承于 Fire 各基类文件）：

A、实体类：Demo.java（对应数据表分别有三个字段：id、name、age）

```
package fire.entity;
/**
 * 实体类:演示
 */
public class Demo extends Entity {

    private static final long serialVersionUID = 1L;
    // 无需id定义，基类里已实现
    private String name;// 姓名
    private int age;  // 年龄

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    // 保存前处理
    @Override
    public void onSave() {

    }
    // 更新前处理
    @Override
    public void onUpdate() {

    }
    @Override
    public String toString() {
        return
this.getClass().getSimpleName()+"[id="+id+",name="+name+",age="+age+"]";
    }
}
```

B、Dao实现类：DemoDaoImpl.java

```
package fire.dao.impl;
import fire.entity.Demo;
/**************************************************************************
* Dao实现类:演示
* @Author JSPGen
* @CreateDate 2013年03月
* @Address www.jspgen.com
* @Email jspgen@163.com QQ 190582560
**************************************************************************/
public class DemoDaoImpl extends DaoImpl<Demo, String> {
    /**
    // 初始化
    public DemoDaoImpl() {
        // 此处做以下定义，将减少dao使用面，建议将以下定义放到service层(同一个dao可多次定义使用，选择不同数据库；若无service层，则需在此类中指定数据连接对象)
```

```
        // 指定访问对象(默认)
        // super.setAccess(FireBuild.getAccess());

        // 指定访问对象：数据源名称, 指定操作表【Map型须设定】
        // super.setAccess(FireBuild.getAccess(""), tableName);

        // 指定访问对象：数据源名称, 指定操作表【Map型须设定】, 表主键名称(类或表字段名), 表主键类型(id、uuid)
        // super.setAccess(FireBuild.getAccess(""), tableName, keyName, keyType);
    }
    */

    // 以下通用方法仅为演示，实际应用中凡通用的方法都可以在业务层直接调用，无需再重写

    // 增加数据(调用通用dao方法)
    public Demo save(Demo demo) {
        return super.save(demo);
    }
    // 增加数据(写SQL语句式)
    public Demo insert(Demo demo) {
        if(Grapes.isEmpty(demo))throw new IllegalArgumentException("entity is required");
        String sql = "insert into " + getTableName() + " (id, title) values (:id, :title)";
        getAccess().createSQL(sql).setParameter(demo).executeUpdate();
        return demo;
    }

    // 查询数据(调用通用dao方法)
    public Demo get(String id) {
        return super.get(id);
    }
    public Demo getLoad(Demo demo) {
        return super.load(demo);
    }

    // 查询数据(写SQL语句式)
    public Demo getDemo(String id) {
        if(Grapes.isEmpty(id))throw new IllegalArgumentException("id is required");
        String sql = "select * from " + getTableName() + " where id='"+ id +"'";
        return getAccess().createSQL(sql).unlist(Demo.class);
```

```
    }

    /**
     * 查询数据，分页列表(调用通用dao方法)
     * @param sql SQL语句(含变量)
     * @param params SQL参数【支持字符索引、数字索引(从0开始)】
     *@param pager 必备参数：pageName(分页文件名)、pageSize(分页大小)、pageNumber(当前页数)
     * @return Pager
     */
    public Pager findPager(String sql, Map<Object, Object> params, Pager pager){
        return super.findPager(sql, params , pager);
    }
/**
// 分页数据显示
Pager pager = DemoDaoImpl.findPager(…);
List<Map<String, Object>> list = (List<Map<String, Object>>)pager.getResults();
if (list != null){
    for(int i=0; i<list.size(); i++){
        Map<String, Object> map = (Map<String, Object>)list.get(i);
    System.out.println("Name:"+map.get("name")+"=="+"Age:"+map.get("age"));
    }
}
// 默认分页栏
System.out.println(pager.getPageBar());
// pager.getPageBar0()、pager.getPageBar1()
// Pager 类使用详情情，可参考Grapes工具包API说明。
*/
    // 其他方法 …
}
```

C、Service 实现类：DemoServiceImpl.java

```
package fire.service.impl;
import fire.FireImpl;
import fire.dao.impl.DemoDaoImpl;
import fire.entity.Demo;
/**********************************************************************
 * Service实现类:演示
 * @Author JSPGen
 * @CreateDate 2013年03月
 * @Address www.jspgen.com
 * @Email jspgen@163.com QQ 190582560
```

```
 ********************************************************************************/
public class DemoServiceImpl extends ServiceImpl<Demo, String> {

    private DemoDaoImpl demoDao;

    // 初始化
    public DemoServiceImpl() {
        demoDao = new DemoDaoImpl();
        // 指定操作对象(默认)
        demoDao.setAccess(FireBuild.getFireAccess());

        // 指定操作对象：数据源名称, 指定操作表【Map型须设定】
        // demoDao.setAccess(FireBuild.getFireAccess(""), tableName);
        // 指定操作对象：数据源名称, 指定操作表【Map型须设定】, 表主键名称(类、表字段名)、表主键类型(id、uuid)
        // demoDao.setAccess(FireBuild.getFireAccess(""), tableName, keyName, keyType);
        super.setDao(demoDao);   // 指定操作Dao
    }
    // 根据页面需要组合Dao实现类中的方法 …
}
```

五、扩展（增加连接池支持）

Fire 默认连接对象支持 BoneCP 连接池与 Proxool 连接池，这两款连接池组件在业界口碑都还不错，我们推荐采用 BoneCP 连接池，如果准备采用除这两款之外的其它连接池组件就需要自行扩展了。

Fire 已帮我们实现了固定接口，我们只需要完成连接池的实现类就可以了，以 BoneCP 连接池实现类为例，文件地址位于 fire.connection.provider 目录下，源码如下：

```
package fire.connection.provider;

import com.jolbox.bonecp.BoneCP;
import com.jolbox.bonecp.BoneCPConfig;
import com.jolbox.bonecp.hooks.ConnectionHook;

import fire.FireException;
import fire.connection.ConnectionProvider;
```

```
import fire.connection.FireConfig;
import grapes.Grapes;

import java.sql.Connection;
import java.sql.Driver;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * 数据库连接对象【连接池】实现(BoneCP)
 *
 * @author JSPGen
 * @copyright (c) JSPGen.com
 * @created 2013年03月
 * @email jspgen@163.com
 * @address www.jspgen.com
 */
public class BoneCPConnectionProvider implements ConnectionProvider {

    // 日志工具
    private static Logger logger =
LoggerFactory.getLogger(BoneCPConnectionProvider.class);
    private String name;     // 配置名称
    private String separator;   // 间隔符
    private String tablePrefix; // 数据表前缀

    private boolean readonly;   // 是否只读
    private Integer isolation;  // 隔离级别
    private boolean autocommit; // 事务自动提交，默认true：每条SQL语句都会按单独事务
提交
    private boolean logSQL;
    private Driver driver;      // 数据库驱动
    private BoneCP pools;
    private BoneCPConfig config;

    /**
     * 连接配置
     * @param props 属性对象
     *
     */
```

```
    public void configure(Properties props) throws FireException {
        try {
            this.config = new BoneCPConfig(props);
            this.name = props.getProperty(FireConfig.CONNECTION_NAME_KEY);
            this.separator =
props.getProperty(FireConfig.CONNECTION_SEPARATOR);
            this.tablePrefix =
props.getProperty(FireConfig.CONNECTION_TABLE_PREFIX);

            String driverName =
props.getProperty(FireConfig.CONNECTION_DRIVER);
            String url = props.getProperty(FireConfig.CONNECTION_URL);
            String username =
props.getProperty(FireConfig.CONNECTION_USERNAME);
            String password =
props.getProperty(FireConfig.CONNECTION_PASSWORD);

            if (Grapes.isNotEmpty(url)) this.config.setJdbcUrl(url);
            if (Grapes.isNotEmpty(username))
    this.config.setUsername(username);
            if (Grapes.isNotEmpty(password))
    this.config.setPassword(password);

            this.readonly =
Grapes.isBoolean(props.getProperty(FireConfig.CONNECTION_READONLY), false);
            this.isolation =
Grapes.isInteger(props.getProperty(FireConfig.CONNECTION_ISOLATION), 4);
            this.autocommit =
Grapes.isBoolean(props.getProperty(FireConfig.CONNECTION_AUTOCOMMIT ), true);
            this.logSQL =
Grapes.isBoolean(props.getProperty(FireConfig.CONNECTION_LOGSQL ), true);

            //logger.debug(this.config.toString());

            // 加载驱动
            if (Grapes.isNotEmpty(driverName)){
                try{
                    Driver driver =
(Driver)Class.forName(driverName).newInstance();
                   DriverManager.registerDriver(driver);  this.driver = driver;
                } catch (ClassNotFoundException e) {
                   logger.warn("Register driver fail: " + driverName); return ;
// 注册驱动失败
                }
```

```
            }
            if (Grapes.isNotEmpty(this.config.getConnectionHookClassName())) {
                Object hookClass =
Grapes.getBean(this.config.getConnectionHookClassName());
                this.config.setConnectionHook((ConnectionHook) hookClass);
            }

            // 创建连接池
            this.pools = createPool(this.config);
        } catch (Exception e) {
            throw new FireException(e);
        }
    }

    /**
     * 获取配置名称
     *
     * @return String
     */
    public String getName(){
        return name;
    }

    /**
     * 获取间隔符(表名、字段与实体类转换需要)
     *
     * @return String
     */
    public String getSeparator(){
        return separator;
    }

    /**
     * 获取表前缀
     *
     * @return String
     */
    public String getTablePrefix(){
        return tablePrefix;
    }

    /**
     * 获取是否记录SQL语句
     *
```

```
     * @return String
     */
    public boolean isLogSQL(){
        return logSQL;
    }

    /**
     * 获取连接
     */
    public Connection getConnection() throws SQLException {
        Connection connection = this.pools.getConnection();
        boolean success = false;
        try {
            if (connection.isReadOnly() != this.readonly) {
                connection.setReadOnly(this.readonly);
            }
            if (connection.getTransactionIsolation() != this.isolation) {
                connection.setTransactionIsolation(this.isolation);
            }
            if (connection.getAutoCommit() != this.autocommit) {
                connection.setAutoCommit(this.autocommit);
            }
            success = true;
            return connection;
        } finally {
            if (!success){
                try {
                    connection.close();
                } catch (Exception e) {
                    logger.warn("Failed to close a connection", e);
                }
                connection = null;
            }
        }
    }

    /**
     * 关闭特定连接
     * @param conn 特定连接
     */
    public void close(Connection conn) throws SQLException {
        conn.close();
    }
```

```
    /**
     * 关闭所有连接(关闭连接池)
     */
    public void close() throws FireException {
        // 关闭连接池
        if (pools != null)   {
            pools.close();
            pools.shutdown();
            pools = null;
        }

        // 撤销注册驱动
        if(driver != null) {
            try {
               DriverManager.deregisterDriver(driver);
            } catch(SQLException e) {
            logger.warn("Deregister drive fail: " + driver.getClass().getName(),
e); // 撤销驱动失败
            } finally {
            driver = null;
            }
        }
    }

    /**
     * 获取配置
     */
    protected BoneCPConfig getConfig() {
        return config;
    }

    /**
     * 创建连接池
     */
    protected BoneCP createPool(BoneCPConfig config) throws SQLException {
        return new BoneCP(config);
    }
}
```

六、关键方法 API

更多调用方法及参数说明，请查看在线 API 文档。

6.1、访问对象可调用方法（FireAccess）：

```
/**
 * 获取数据库连接
 * @return Connection
 */
public Connection getConnection();

/**
 * 获取数据库类型
 * @return String
 */
public String getDatabaseGenre();

/**
 * 获取数据库间隔符
 * @return String
 */
public String getSeparator();

/**
 * 获取数据库表前缀
 * @return String
 */
public String getTablePrefix();

/**
 * 获取数据库表主键名称
 * @return String
 */
public String getKeyName();

/**
 * 获取数据库表主键值类型[id uuid]
 * @return String
 */
public String getKeyValueType();
```

```java
/**
 * 获取当前连接运行过的SQL语句[参数]集合
 * @return ArrayList<Map<String, Object>>
 */
public List<Map<String, Object>> getSQLParameters();

/**
 * 打开事务
 */
public FireAccess begin() throws FireException;

/**
 * 事务是否打开
 * @return boolean
 */
public boolean isBegin() throws FireException;

/**
 * 事务提交
 * @param boolean autoCommit 恢复自动提交(默认：是)
 * @return void
 */
public void commit(boolean autoCommit) throws FireException;
public void commit() throws FireException;

/**
 * 事务回滚
 * @return void
 */
public void rollback() throws FireException;

/**
 * 关闭连接
 */
public void close() throws FireException;

/**
 * 清空(关闭)连接池
 */
public void clear() throws FireException;
/**
 * 连接是否关闭
 * @return boolean
 */
```

```
    public boolean isClose() throws FireException;

    /**
     * 获取某表主键Id值
     * @param table Id统计表表名称(默认：id)
     * @param name 某表名称
     * @return long
     */
    public String getId(String table, String name) throws FireException;
    public String getId(String name) throws FireException;

    /**
     * 创建SQL语句
     * @param sql SQL语句
     */
    public FireAccess createSQL(String sql) throws FireException;

    /**
     * 参数赋值(单个)
     * @param position 参数索引(从0开始)
     * @param value 参数值
     */
    public FireAccess setParameter(int position, Object value) throws
FireException;

    /**
     * 参数赋值(单个)
     * @param name 参数索引(字符索引、字符数字索引)
     * @param value 参数值
     */
    public FireAccess setParameter(String name, Object value) throws
FireException;
    public FireAccess setParameter(Map<?, ?> paramMap) throws FireException;
    public FireAccess setParameter(Object obj) throws FireException;

    /**
     * 参数赋值(批量)
     * @param sql SQL语句(参数：?符、字符变量符)
     * @param prams 参数集合(参数：数字、字符数字、字符变量)
     */
    public FireAccess setSQLParameter(String sql, Map<?, ?> prams) throws
FireException;
    public FireAccess setSQLParameter(String sql, Object obj) throws
FireException;
```

```
    /**
     * 更新(封装式)
     * 注：执行完毕后会清空最近一次创建的SQL语句
     * @return boolean
     */
    public String executeUpdate() throws FireException;

    /**
     * 更新(拼接式)
     * 若开启事务则之前所定义参数 setParameter 失效
     * @param sql SQL语句
     * @throws SQLException
     * @return int 影响行数
     */
    public String executeUpdate(String sql) throws FireException;

    /**
     * 更新(传参式)
     * @param sql SQL语句(参数：?符)
     * @param prams 参数集合(从0开始，优先级高于setParameter所定义参数)
     * @param isflag 是否返回主键(默认：false，反回影响行数，若开启事务则此处自动为否)
     * @return String 影响行数(主键)
     */
    public String executeUpdate(String sql, Map<Integer, Object> prams, boolean
isflag) throws FireException;
    public String executeUpdate(String sql, Map<Integer, Object> prams) throws
FireException;

    /**
     * 查询(封装式)
     * 注：执行完毕后会清空最近一次创建的SQL语句，故不可与last、list同时使用(last、list
中有此方法)，可返回结果集
     * @return ResultSet
     */
    public ResultSet executeQuery() throws FireException;

    /**
     * 查询(拼接式)
     * @param sql SQL语句
     * @return ResultSet
     */
    public ResultSet executeQuery(String sql) throws FireException;
```

```
    /**
     * 查询(传参式)
     * @param sql SQL语句(参数：?字符)
     * @param prams 参数集合(从0开始，优先级高于setParameter所定义参数)
     * @return ResultSet
     */
    public ResultSet executeQuery(String sql, Map<Integer, Object> prams) throws
FireException;


    /**
     * 获取SQL查询结果集
     * @param rs 记录结果集[可配合executeQuery(sql)操作](默认：获取自身查询)
     * @param isflag 调用完毕后关闭结果集(默认：true)
     * @return List<Map<String, Object>>
     */
    public List<Map<String, Object>> list(ResultSet rs, boolean isflag);
    public List<Map<String, Object>> list();
    public Map<String, Object> unlist(ResultSet rs, boolean isflag);
    public Map<String, Object> unlist();

    /**
     * 获取SQL查询结果集
     * @param clazz 实体类对象
     * @param rs 记录结果集[可配合executeQuery(sql)操作](默认：获取自身查询)
     * @param isflag 调用完毕后关闭结果集(默认：true)
     * @return List<T>
     */
    public <T> List<T> list(Class<T> clazz, ResultSet rs, boolean isflag);
    public <T> List<T> list(Class<T> clazz);
    public <T> T unlist(Class<T> clazz, ResultSet rs, boolean isflag);
    public <T> T unlist(Class<T> clazz);

    /**
     * 获取SQL查询结果集(某一列)
     * @param int index 获取列值，从1开始
     * @param rs 记录结果集[可配合executeQuery(sql)操作](默认：获取自身查询)
     * @param isflag 调用完毕后关闭结果集(默认：true)
     * @return List<Object>
     */
    public List<Object> result(int index, ResultSet rs, boolean isflag);
    public List<Object> result(int index);
    public Object unresult(int index, ResultSet rs, boolean isflag);
    public Object unresult(int index);
```

```
public List<Object> result(ResultSet rs, boolean isflag);
public List<Object> result();
public Object unresult(ResultSet rs, boolean isflag);
public Object unresult();

/**
 * 获取SQL查询结果集总数(直接跳到最后一条记录)
 * @param rs 记录结果集[可配合executeQuery(sql)操作](默认：获取自身查询)
 * @param isflag 调用完毕后关闭结果集(默认：true)
 * @return long
 */
public long last(ResultSet rs, boolean isflag);
public long last(ResultSet rs);
public long last(boolean isflag);
public long last();
```

6.2、基类 Dao 文件可调用方法（含 CRUD 通用方法）：

其中，Map 型基类 Dao 文件于此类似仅参数类型由实体类变更为 Map 型即可。

```
/**
 * 设置数据访问对象
 */
public void setAccess(String[] names);
public void setAccess(String name);
public void setAccess(FireAccess access);

/**
 * 设置表名称
 */
public void setTableName(String name);

/**
 * 关闭数据访问对象连接
 */
public void close();

/**
 * 清空(关闭)数据对象连接池
 */
public void clear();

/**
```

```
     * 获取信息
     * @param entity 实体类
     * @return T
     */
    public T load(T entity);

    /**
     * 获取信息
     * @param id 信息主键Id
     * @return T
     */
    public T get(PK id);

    /**
     * 获取信息
     * @param key 主键名称
     * @param value 主键值
     * @return T
     */
    public T get(String key, String value);

    /**
     * 获取列表
     * @param sort 排序字段名称
     * @param order 排序顺序(默认降序，asc升序、desc降序)
     * @return List<T>
     */
    public List<T> getList(String sort, String order);

    /**
     * 获取总数
     * @return Long
     */
    public Long getCount();

    /**
     * 保存信息
     * @return boolean
     */
    public T save(T entity);

    /**
     * 更新信息
     * @param entity 实体类
```

```
     * @return boolean
     */
    public void update(T entity);

    /**
     * 删除信息
     * @param entity 实体类型参数
     * @param PK 主键Id
     */
    public void delete(T entity);
    public void delete(PK id);
    public void delete(PK[] ids);
    public void delete(String key, String value);
    public void delete(String key, String[] values);

    /**
     * 查找分页
     * @param sql SQL语句(含变量)
     * @param params SQL参数【支持字符索引、数字索引(从0开始)】
     * @param pager 必备参数：pageName(分页文件名)、pageSize(分页大小)、pageNumber(当前页数)
     * @return Pager
     */
    public Pager findPager(String sql, Map<Object, Object> params, Pager pager);
```

七、附

7.1、获取数据主键思路

在 Fire 体系中，添加数据操作时有两种方法可获得主键(若采用 Fire 基类，则可以在 Dao 或 Service 实现类中配置主键名称及主键类型，即可实现自动填充主键值功能)：

A、采用 uuid 字符串：Grapes.uuid(boolean isflag)；// 参数为是否含中划线，默认为否

B、采用整型序列数字：调用 FireAccess 访问对象中的 getId(String name)方法，来获取该数据库中某表当前最大 Id 值，提前需要在当前数据库中设计如下数据表：

```
DROP TABLE IF EXISTS `id`;
CREATE TABLE `id` (
```

```
    `id` bigint(13) NOT NULL COMMENT 'Id',
    `name` varchar(50) NOT NULL COMMENT '名称',
    PRIMARY KEY (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='Id 表';
```

7.2、数据表转实体类工具

执行文件位于 fire.sql.tools 目录，MySQLBean.class 文件，调用生成代码：

```
public static void main(String[] args) {
    String name = "";           // 数据源名称
    String dir = "D:/bean";     // 保存目录
    String packName = "jspgen"; // 包名

    MySQLBean mb = new MySQLBean();
    /**
     * 生成某数据源所有表实体类
     * @param name          数据源别名
     * @param packName      包名
     * @param tableName     数据表(为空则所有表)
     * @param tableComment  数据表备注(以“;”结尾)
     * @param ignore        可忽视的方法名
     * @param dir           保存目录
     * @return void
     */
    // mb.parseTable(name, packName, tableName, tableComment, dir);
    // 生成某数据源所有表实体类
    mb.parseTable(name, packName, new String[]{"id"}, dir);
}
```