# mango

**mango的中文名是“芒果”，它是一个极速分布式ORM框架。**

**特性:**

* 超高性能，响应速度接近直接使用JDBC
* 采用接口与注解的形式定义DAO，完美结合db与cache操作
* 支持动态sql，可以构造任意复杂的sql语句
* 支持多数据源，分表，分库，事务
* 提供拦截器功能，利用拦截器可为mango框架扩展各种自定义插件
* 独创“函数式调用”功能，能将任意复杂的对象，映射到数据库的表中
* 高效详细的实时统计系统，方便开发者随时了解自己的系统
* 独立jar包，不依赖其它jar包
* 提供便捷的spring插件，与spring无缝集成

添加依赖：

**<dependency>**

**<groupId>**org.jfaster**</groupId>**

**<artifactId>**mango**</artifactId>**

**<version>**1.6.1**</version>**

**</dependency>**

## 构造数据源并初始化mango对象

mango框架对java标准数据源 javax.sql.DataSource 进行了简单实现，所以这里构造数据源不需要引入第三方jar包。

初始化数据源需要4个参数:

* **driverClassName**: 驱动程序类名，这里我们使用MySQL驱动，所以类名是 com.mysql.jdbc.Driver 。
* **url**: 连接数据库的url，这里我们将连接到本地MySQL的mango\_example库，所以地址为 jdbc:mysql://localhost:3306/mango\_example 。
* **username**: 数据库用户名，这里我们使用root作为用户名。
* **password**: 用户名所对应的密码，这里我们使用root作为密码。

**注解说明：**

（1）@DB(table="表名") 或@DB 全名为

**@org.jfaster.mango.annotation.DB** ，dao接口必须使用它来修饰，这样这个dao接口才能被mango框架接受。

（2）@SQL("增删改查SQL语句") 注解的全名为

@org.jfaster.mango.annotation.SQL ，它被用来修饰下面对应的方法。

（3）@ReturnGeneratedId 用于修饰方法的返回值，注解 的全名为@org.jfaster.mango.annotation.ReturnGeneratedId ，被修饰的方法具有以下两种返回值类型：

int或java.lang.Integer：返回int类型的自增idlong或java.lang.Long：返回long类型的自增id

（4）@Rename 对参数进行重命名

（5）@Results/@Result 注解 @org.jfaster.mango.annotation.Results 和注解 @org.jfaster.mango.annotation.Result 来完成自定义映射规则匹配。

（6）@Mapper 定义映射类

（7）@ID 指定表的自增主键

（8）@ID(autoGenerateId = false) 指定表的非自增主键

（9）@column("uid")定义映射字段，将类A的userId属性映射到表的uid字段中

private Integer userId

（10）@Ignore 表中没有这个字段，可以使用该注解忽略这个属性

（11）@DB(name="") 用于指定所使用的数据源工厂的名字

（12）@Sharding(tableShardingStrategy= xxx.class) 用于将表分片策略与DAO接口进行绑定

（13）@TableShardingBy("") 用于指定给表分片策略传入的参数。

（14）@Sharding(databaseShardingStrategy = xxx.class) 用于将数据库分片策略与DAO接口进行绑定

（15）@DatabaseShardingBy("") 用于指定给数据库分片策略传入参数。

（16）@ShardingBy 使用@ShardingBy注解等于同时使用@DarabaseShardingBy注解与@TableShardingBy注解

（17）@Getter(IntegerListToStringFunction.class) 用于将整型列表转化为字符串

@Getter(IntegerListToStringFunction.class)

public List<Integer> getStudentIds() {

return studentIds;

}

@Getter注解的全名为@org.jfaster.mango.annotation.Getter，它使用在get方法上。

IntegerListToStringFunction类的全名为org.jfaster.mango.invoker.function.IntegerListToStringFunction，它用于将整型列表转化为字符串。

（18）@Setter(StringToIntegerListFunction.class) 用于将字符串转化为整型列表

@Setter(StringToIntegerListFunction.class)

public void setStudentIds(List<Integer> studentIds) {

this.studentIds = studentIds;

}

@Setter注解的全名为@org.jfaster.mango.annotation.Setter，它使用在set方法上。

StringToIntegerListFunction类的全名为org.jfaster.mango.invoker.function.StringToIntegerListFunction，它用于将字符串转化为整型列表。

（19）@Getter(EnumToIntegerFunction.class) 调用枚举对象的ordinal方法，将枚举对象转化为数字

@Getter(EnumToIntegerFunction.class)

public Gender getGender() {

return gender;

}

EnumToIntegerFunction类的全名为org.jfaster.mango.invoker.function.enums.EnumToIntegerFunction。

（20）@Setter(IntegerToEnumFunction.class) 用于将数字转化为枚举对象

@Setter(IntegerToEnumFunction.class)

public void setGender(Gender gender) {

this.gender = gender;

}

IntegerToEnumFunction类的全名为org.jfaster.mango.invoker.function.enums.IntegerToEnumFunction。

（21）Getter(ObjectToGsonFunction.class) ObjectToGsonFunction类的全名为org.jfaster.mango.invoker.function.json.ObjectToGsonFunction，它使用gson包，能将任意对象转化为json字符串。

@Getter(ObjectToGsonFunction.class)

public SubCard getSubCard() {

return subCard;

}

（22）Setter(GsonToObjectFunction.class) GsonToObjectFunction类的全名为org.jfaster.mango.invoker.function.json.GsonToObjectFunction，它使用gson包，能将json字符串转化为任意对象。

@Setter(GsonToObjectFunction.class)

public void setSubCard(SubCard subCard) {

this.subCard = subCard;

}

（23）@Cache(prefix="user", expire=Hour.class, num = 2) 表示需要使用缓存，参数prefix表示key前缀，比如说传入uid=1，那么缓存中的key就等于user\_1，参数expire表示缓存过期时间，Hour.class表示小时，配合后面的参数num＝2表示缓存过期的时间为2小时。

（24）@CacheBy用于修饰key后缀参数，在delete，update，getUser方法中@CacheBy都是修饰的uid，所以当传入uid=1时，缓存中的key就等于user\_1。

（25）@CacheIgnored表示该方法不需要缓存。需要注意的是，如果使用了@Cache注解，@CacheBy和@CacheIgnored二者必须有一个存在。

@DB

@Cache(prefix = "user", expire = Hour.class, num = 2)

public interface SingleKeySingeValueDao {

@CacheIgnored

@SQL("insert into user(uid, name) values(:1, :2)")

public int insert(int uid, String name);

@SQL("delete from user where uid=:1")

public int delete(@CacheBy int uid);

@SQL("update user set name=:2 where uid=:1")

public int update(@CacheBy int uid, String name);

@SQL("select uid, name from user where uid=:1")

public User getUser(@CacheBy int uid);

}

**更新**

更新主要包含insert，delete与update这三种操作。

通常情况化更新操作支持四种类型的返回值：

1、void或java.lang.Void：不返回值

2、int或java.lang.Integer：返回有多少行数据受到了影响

3、long或java.lang.Long：返回有多少行数据受到了影响

4、boolean或java.lang.Boolean：false表示没有数据受到影响，true表示有一到多行数据受到影响

**批量更新**

批量更新主要包含insert，delete与update这三种操作。

批量更新的输入只能有一个参数，参数的类型必须是List或Set或Array。

批量更新的输出支持三种类型的返回值：

1、void或java.lang.Void：不返回值

2、int或java.lang.Integer：返回累计有多少行数据受到了影响

3、int[]或java.lang.Integer[]：返回每条更新语句影响到了多少行数据

**Mongo参数绑定**

参数绑定指的是：将接口参数绑定到SQL指定的位置中，也即向SQL中传入参数。

**序号绑定**

序号绑定指的是将接口参数的序号绑定到SQL指定的位置中。 参数的序号从1开始，:1表示使用第1个参数，:2表示使用第2个参数，以此类推。 下面是序号绑定的实例：

@SQL("insert into binding\_user(uid, name, age) values(:1, :2, :3)")

public void addUserByIndex(int uid, String name, int age);

**重命名绑定**

可以使用注解@Rename对参数进行重命名绑定，实例：

@SQL("insert into user(name, age, gender, money, update\_time) values(:name, :age, :gender, :money, :updateTime)")

public void addUserByRename(@Rename("name")String name, @Rename("age")int age, @Rename("gender")boolean gender, @Rename("money")long money, @Rename("updateTime")Date updateTime);

**列表参数绑定**

在SQL中使用in操作的时候，我们会使用到列表参数绑定。 下面是列表参数绑定的实例：

@SQL("select id, name, age from user where id in (:1)")

public List<User> getUsersByIds(List<Integer> ids);

需要注意的是， in (:1) 中的参数必须是List或Set或Array，同时返回参数也必须是List或Set或Array。

**属性绑定**

当接口参数传入的是自定义对象时，我们可以使用属性绑定。 下面是属性绑定的实例：

@SQL("insert into user(id, name, age) values(:1.id, :1.name, :1.age)")

public void addUserByObjIndex(User user);

需要注意的是，User类的属性必须具有get方法， 因为:1.uid将调用getUid()方法获取参数，:1.name将调用getName()方法获取参数，:1.age将调用getAge()方法获取参数。

**属性自动匹配**

在使用自定义对象时，使用:1.uid或:u.uid绑定参数会显得不够简练，mango实现了属性自动匹配功能，使SQL更加简练。 下面是属性自动匹配的实例：

@SQL("insert into user(id, name, age) values(:id, :name, :age)")

public void addUserByProperty(User user);

**混合绑定**

各种参数绑定混合使用：

@SQL("insert into user(id, name, age) values(:id, :name, :age)")

public void addUserByMix(@Rename("myid") int id, User user);

**Mango查询映射**

查询映射指的是：将使用select语句从数据库表中查询到的字段，映射到原生对象或者自定义对象的属性中。

映射到原生对象

当我们通过select语句只查询数据库表中的一列字段时，我们可以将该列字段的数据映射到原生对象中。

所谓原生对象指的是：String，int等基本对象，它们能和数据库中的varchar，int等类型一一对应。

下面是映射到原生对象的实例：

@SQL("select name from user where id = :1")

public String getNameById(int id);

@SQL("select id from user limit :1")

public List<Integer> getIdsLimit(int limit);

映射到自定义对象的属性中：

当我们通过select语句查询数据库表中的多列字段时，我们可以将这些字段的数据映射到自定义对象的属性中。

默认映射规则：

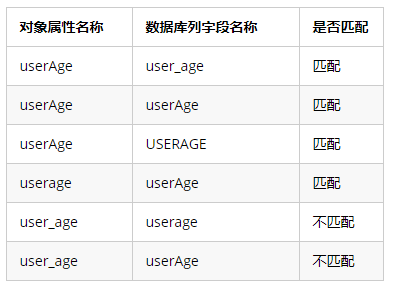
当列字段名称与对象属性名称满足下面的匹配规则时，该列字段的数据将会映射到自定义对象的该属性中：

1、将对象属性名称记做P

2、将P由驼峰式规则变为下划线规则，得到新的名称Q（如：P=”userId”则Q=”user\_id”）

3、如果某个列字段名称与名称P或Q相等（忽略大小写），则匹配满足，能完成映射

匹配规则表：



（一）自定义映射规则

一般来说，我们可以使用默认映射规则完成绝大部分匹配。如果遇到默认映射规则无法完成的匹配，我们可以使用自定义映射规则。

我们可以用使用注解 @org.jfaster.mango.annotation.Results 和注解 @org.jfaster.mango.annotation.Result 来完成自定义映射规则匹配。

@Results({

@Result(column = "id", property = "userId"),

@Result(column = "name", property = "userName")

})

@SQL("select id, name, user\_age, update\_time from mapping\_user where id = :1")

public MappingUser2 getMappingUser2ById(int id);

@Results({

@Result(column = "id", property = "userId"),

@Result(column = "name", property = "userName")

})

@SQL("select id, name, user\_age, update\_time from mapping\_user where id in (:1)")

public List<MappingUser2> getMappingUsers2ById(List<Integer> ids);

（二）手动映射

无论是默认映射规则还是自定义映射规则都是通过反射的形式进行列字段到对象属性的映射。 mango提供了抽象类 org.jfaster.mango.jdbc.AbstractRowMapper，继承该类可以实现手动映射。

**Mango增删改查NOSQL**

CrudDao接口

为实现增删改查NoSQL，mango框架对开发人员提供了 CrudDao 接口，我们只需简单继承CrudDao接口，不需要书写任何SQL，即可获得常用的增删改查方法。

CrudDao定义如下：

public interface CrudDao<T, ID> extends Generic<T, ID> {

void add(T entity);

int addAndReturnGeneratedId(T entity);

void add(Collection<T> entities);

T getOne(ID id);

List<T> getMulti(List<ID> ids);

List<T> getAll();

long count();

int update(T entity);

int[] update(Collection<T> entities);

int delete(ID id);

}

mango框架提供使用方法名的方式进行自定义操作：

方法名以getBy,findBy,queryBy,selectBy开头表示查询

方法名以countBy开头表示计数

方法名以deleteBy,removeBy开头表示删除

以 findById 为例，findBy后面的关键字为Id，表示根据id查询，findById会被转化为SQL：select id, uid, status from #table where id = :1

以 findByIdAndUid 为例，findBy后面的关键字为IdAndUid，表示根据id和uid查询，findByIdAndUid会被转化为SQL：select id, uid, status from #table where id = :1 and uid = :2

以 countByUid 为例，countBy后面的关键字为Uid，表示根据uid计数，countByUid会被转化为SQL：select count(1) from #table where uid = :1

以 deleteByUid 为例，deleteBy后面的关键字为Uid，表示根据uid删除，deleteByUid会被转化为SQL：delete from #table where uid = :1

常用的关键字-SQL对应表

| **关键字** | **样例** | **对应SQL** |
| --- | --- | --- |
| And | findByIdAndName | … where id = :1 and name = :2 |
| Or | findByIdOrName | … where id = :1 or name = :2 |
| Equals | findById,findByIdEquals | … where id = :1 |
| Between | findByStartDateBetween | … where startDate between :1 and :2 |
| LessThan | findByAgeLessThan | … where age < :1 |
| LessThanEqual | findByAgeLessThanEqual | … where age <= :1 |
| GreaterThan | findByAgeGreaterThan | … where age > :1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where age >= :1 |
| IsNull | findByAgeIsNull | … where age is null |
| NotNull | findByAgeNotNull | … where age not null |
| OrderBy | findByAgeOrderByIdDesc | … where age = :1 order by id desc |
| Not | findByLastnameNot | … where lastname <> :1 |
| In | findByAgeIn(Collection<Age> ages) | … where age in (:1) |
| NotIn | findByAgeNotIn(Collection<Age> ages) | … where age not in (:1) |
| True | findByActiveTrue | … where active = true |
| False | findByActiveFalse | … where active = false |

带分页的自定义数据查询

分页查询的代码如下：

@DB(table = "t\_order")

public interface OrderPageNoSqlDao extends CrudDao<Order, Integer> {

List<Order> findByUid(int uid, Page page);

List<Order> findByIdOrUid(int id, int uid, Page page);

}

**Mango数据源工厂**

一般情况下，应用服务（或WEB服务）连数据库有3种大的方式：

1、应用服务连单一数据库，所有的数据库读写请求都使用该数据库库，适用于小规模系统

2、应用服务连主从数据库，数据库写请求使用主库，数据库读请求使用从库，适用于中等规模系统

3、应用服务连多个单一数据库，多个主从数据库组成的混合数据库集群，适用于大规模系统

**单一数据库**

应用服务所有的数据库读写请求使用单一数据库时，我们使用简单数据源工厂SimpleDataSourceFactory管理数据源。

实例代码：

String url = "jdbc:mysql://localhost:3306/mango\_example";

String username = "root"; // 用户名

String password = "root"; // 密码

//初始化数据源datasource

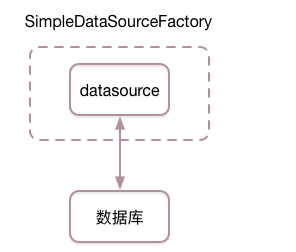
DataSource datasource = new DriverManagerDataSource(driverClassName, url, username, password);

//初始化数据源工厂

DataSourceFactory dsf = new SimpleDataSourceFactory(datasource);

//使用数据源工厂初始化mango对象

Mango mango = Mango.newInstance(dsf);



**主从数据库**

应用服务的数据库写请求使用主库，数据库读请求使用从库时，我们使用主从数据源工厂MasterSlaveDataSourceFactory管理数据源。

实例代码：

String driverClassName = "com.mysql.jdbc.Driver";

String url = "jdbc:mysql://localhost:3306/mango\_example";

String username = "root"; // 用户名

String password = "root"; // 密码

//初始化数据源master，用于连接主库

DataSource master = new DriverManagerDataSource(driverClassName, url, username, password);

//初始化数据源slave1和slave2用于连接从库，为了简单，参数与主库一致，实际情况下从库有不同的url，username，password

DataSource slave1 = new DriverManagerDataSource(driverClassName, url, username, password);

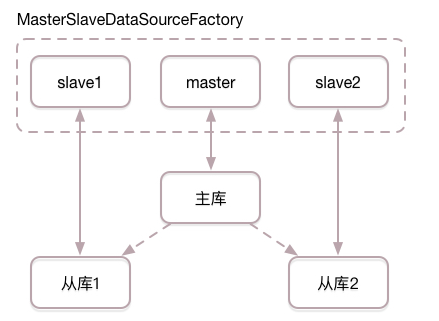
DataSource slave2 = new DriverManagerDataSource(driverClassName, url, username, password);

//初始化数据源工厂

DataSourceFactory dsf = new MasterSlaveDataSourceFactory(master, Arrays.asList(slave1, slave2));

//使用数据源工厂初始化mango对象

Mango mango = Mango.newInstance(dsf);



**混合数据库集群**

当应用服务需要连多个单一数据库或多个主从数据库组成的混合数据库集群时，我们可以使用多个数据源工厂管理数据源，并使用多个数据源工厂来初始化。

String driverClassName = "com.mysql.jdbc.Driver";

String username = "root"; // 用户名

String password = "root"; // 密码

// 主从数据源工厂，为了简单，从库参数与主库一致

String url1 = "jdbc:mysql://localhost:3306/mango\_example\_db1";

String name1 = "dsf1";

DataSource master = new DriverManagerDataSource(driverClassName, url1, username, password);

DataSource slave1 = new DriverManagerDataSource(driverClassName, url1, username, password);

DataSource slave2 = new DriverManagerDataSource(driverClassName, url1, username, password);

List<DataSource> slaves = Arrays.asList(slave1, slave2);

//初始化主从数据源工厂

DataSourceFactory dsf1 = new MasterSlaveDataSourceFactory(name1, master, slaves);

// 简单数据源工厂

String name2 = "dsf2";

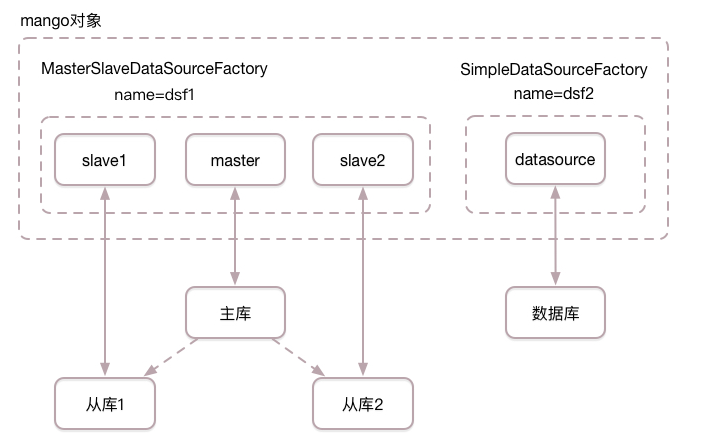
String url2 = "jdbc:mysql://localhost:3306/mango\_example\_db2";

DataSource datasource = new DriverManagerDataSource(driverClassName, url2, username, password);

//初始化单一数据库工厂

DataSourceFactory dsf2 = new SimpleDataSourceFactory(name2, datasource);

Mango mango = Mango.newInstance(Arrays.asList(dsf1, dsf2));



**Mango表分片**

表分片通常也被称为分表，散表。

当某张表的数据量很大时，sql执行效率都会变低，这时通常会把大表拆分成多个小表，以提高sql执行效率。 我们将这种大表拆分成多个小表的策略称之为表分片。

要实现表分片，需要定义一个表分片类，这个类要实现 TableShardingStrategy 接口中的 getTargetTable 方法。

getTargetTable方法是表分片策略的核心，共两个输入参数，输出则为最终需要访问的表名字，所以我们通过实现getTargetTable方法计算最终需要访问的表名字。

第1个参数table为@DB注解中table参数所定义的全局表名。

第2个参数是自定义传入的参数，这里我们使用uid计算如何分表，所以第2个参数是uid。

需要注意的是，第2个参数是一个泛型参数，这里由于uid是整形数字，所以类型定义为Integer。

**Mango数据库分片**

数据库分片通常也被称为分库，散库等。

当我们在某个库中，把某张大表拆分成多个小表后还不能满足性能要求，这时我们需要把一部分拆分的表挪到另外一个库中，以提高sql执行效率。

要实现数据库分片，需要定义一个数据库分片类，这个类要实现DatabaseShardingStrategy 接口中的 getDataSourceFactoryName 方法。

getDataSourceFactoryName方法是数据库分片策略的核心，返回最终请求的数据源工厂名称。

而getDataSourceFactoryName方法的输入参数是一个自定义传入的参数，这里由于我们要使用uid计算如何分库，所以参数为uid。

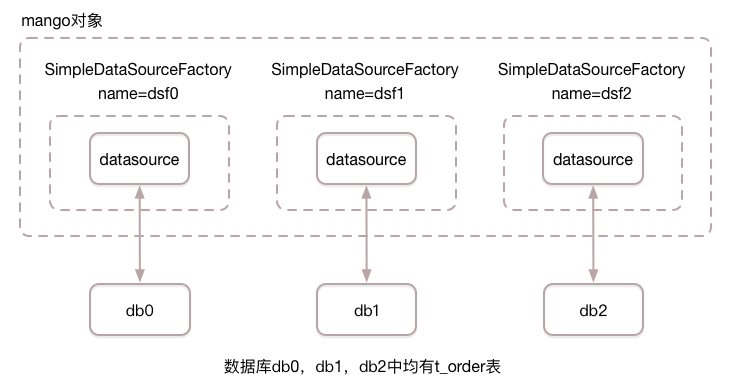
需要注意的是，输入参数是一个泛型参数，这里由于uid是整形数字，所以类型定义为Integer。

我们使用 @Sharding 注解中的databaseShardingStrategy参数，将数据库分片策略与DAO接口进行绑定。

引入了 @ShardingStrategy 接口，实现@ShardingStrategy接口等与同时实现@DatabaseShardingStrategy接口与@TableShardingStrategy接口。。

一维度分片策略：使用一个字段属性作为分片策略的计算参数；

多维度分片策略：同时使用多个分片策略计算参数，不同的方法指定不同的分片策略；



如果想要同时实现分表和分库功能，需要实现ShardingStrategy接口。

ShardingStrategy接口同时继承DatabaseShardingStrategy, TableShardingStrategy接口，然后分别实现getDataSourceFactoryName方法和getTargetTable方法即可。

**Mango函数式声明**

**我们需要将java中的列表，集合，自定义类映射到关系型数据库的某个字段中，直接插入取出显然是不行的，这时我们就需要使用函数式调用功能，将这些数据库不能识别的类型转化为数据库能够识别的类型插入，后续取出时再通过函数式调用转化为java中的复杂类型。**

**（1）列表与字符串互转**

**@Getter(IntegerListToStringFunction.class) 将整型列表转化为字符串**

**@Setter(StringToIntegerListFunction.class) 将字符串转化为整型列表**

**（2）枚举与数字互转**

**@Getter(EnumToIntegerFunction.class) 将枚举对象转化为数字**

**@Setter(IntegerToEnumFunction.class) 将数字转化为枚举对象**

**（3）复杂类与字符串互转**

**@Getter(ObjectToGsonFunction.class) 将任意对象转化为json字符串**

**@Setter(GsonToObjectFunction.class) 将json字符串转化为任意对象**

**Mango拦截器**

拦截器是mango框架中一个非常重要的功能，它为mango框架提供了扩展插件的可能。

当我们为mango框架添加拦截器后，最终执行的SQL会依次通过这些拦截器后再被执行。 所以我们可以通过拦截器对最终执行的SQL进行各种操作，如修改SQL，记录SQL等。

通过自定义拦截器，我们可以为mango框架扩展出许许多多的插件，最为常用的 分页查询 插件就可以通过拦截器来实现。

mango框架将拦截器分为了两类：查询拦截器与更新拦截器。

**查询拦截器**：所有查询请求最终需要执行的SQL都需要依次通过查询拦截器处理。

要实现一个自己的查询拦截器，需要定义一个类(MyQueryInterceptor)，并继承 QueryInterceptor 类；

在初始化mango对象时，需要调用addInterceptor方法将我们的拦截器MyQueryInterceptor添加到了mango对象中。

public class MyQueryInterceptor extends QueryInterceptor {

public void interceptQuery(

BoundSql boundSql,

List<Parameter> parameters,

DataSource dataSource) {

System.out.println("sql: " + boundSql.getSql());

System.out.println("args: " + boundSql.getArgs());

}

}

QueryInterceptor中的interceptQuery方法有3个输入参数：

1、BoundSql封装了最终将要被执行的SQL

2、List<Parameter>封装了被执行方法的参数列表信息

3、DataSource则是SQL执行时所使用的数据源

**更新拦截器**：所有更新请求最终需要执行的SQL都需要依次通过更新拦截器处理。

要实现一个自己的更新拦截器，需要定义一个类(MyUpdateInterceptor)，并继承继承 UpdateInterceptor 类。

在初始化mango对象时，我们调用addInterceptor方法将我们的拦截器(MyUpdateInterceptor)添加到了mango对象中。

public class MyUpdateInterceptor extends UpdateInterceptor {

public void interceptUpdate(

BoundSql boundSql,

List<Parameter> parameters,

SQLType sqlType,

DataSource dataSource) {

System.out.println("sql: " + boundSql.getSql());

System.out.println("args: " + boundSql.getArgs());

}

}

UpdateInterceptor中的interceptUpdate方法有4个输入参数：

1、BoundSql封装了最终将要被执行的SQL

2、List<Parameter>封装了被执行方法的参数列表信息

3、SQLType封装了被执行SQL的类型

4、DataSource则是SQL执行时所使用的数据源

**Mango动态SQL**

**字符串连接**

字符串连接的语法为 #{参数}，请看下面的代码:

@DB

public interface UserDao {

@SQL("select uid, name from user order by #{:1}")

public List<User> getUsers(String orderBy);

}

当orderBy传入uid时，sql被解析为：select uid, name from user order by uid

当orderBy传入name时，sql被解析为：select uid, name from user order by name

**if语句**

if语句主要存在两种形式，[xxx]表示xxx不出现或出现一次：

不包含#elseif，语法为 #if(表达式) 字符串 [ #else 字符串 ] #end

包含#elseif，语法为 #if(表达式) 字符串 #elseif(表达式) 字符串 [ #else 字符串 ] #end，这里的#elseif可以多个

不包含#elseif

@DB

public interface UserDao {

@SQL("select uid, name from user where #if(:1>0) uid=:1 #else uid=-1 #end")

public User getUser(int uid);

}

当uid传入10时，将执行：select uid, name from user where uid = 10

当uid传入0时，将执行：select uid, name from user where uid = -1

包含#elseif

@DB

public interface UserDao {

@SQL("select uid, name from user where #if(:1>0) uid=:1 #elseif(:1==0) uid=1 #else uid=-1 #end")

public User getUser2(int uid);

}

当uid传入10时，将执行：select uid, name from user where uid = 10

当uid传入0时，将执行：select uid, name from user where uid = 1

当uid传入-5时，将执行：select uid, name from user where uid = -1

**Mango集成cache**

mango自身不依赖任何缓存工具，mango对外提供SimpleCacheHandler抽象类，您只需继承SimpleCacheHandler抽象类，

并在实现其中的缓存操作代码（memcached，redis，直接内存等均可），就能享受mango带来的缓存操作便利。

CacheHandler抽象类一共有4个需要实现的抽象方法，它们分别对应着封装缓存的操作:

(1)Object get(String key, Type type) ，根据单个key值从缓存中查找数据，type为返回对象的类型

(2)Map<String, Object> getBulk(Set<String> keys, Type type) ，根据多个key值从缓存中查找数据，返回key-value对应的map，type为map中value对象的类型

(3)void set(String key, Object value, int exptimeSeconds)，向缓存中设置数据，其中exptimeSeconds为缓存失效时间，单位为秒。

(4)void delete(String key) ，根据单个key值从缓存中删除数据。

初始化mango对象

DataSource ds = new DriverManagerDataSource(driverClassName, url, username, password);

Mango mango = Mango.newInstance(ds);

mango.setCacheHandler(new MockRedisHandler());

正常初始化mango对象后，只需要通过setCacheHandler方法传入一个实现了CacheHandler接口的对象即可，

这里我们使用的是自定义的Redis实现的MockRedisHandler。

@DB

@Cache(prefix = "user", expire = Hour.class, num = 2)

public interface UserCacheDao {

@CacheIgnored //不使用缓存

@SQL("insert into user(name, age, gender, money, update\_time) values(:name, :age, :gender, :money, :updateTime)")

public int insert(User user);

@SQL("insert into user(name, age, gender, money, update\_time) values(:1.name, :1.age, :1.gender, :1.money, :1.updateTime)")

public int insertUser(@CacheBy("name") User user); //使用User对象的name属性作为key后缀

@SQL("delete from user where id =:1")

public int delete(@CacheBy int id );

@SQL("update user set name = :2 where id =:1")

public int update(@CacheBy int id, String name);

@SQL("select \* from user where id=:1")

public User getUser(@CacheBy int id);

@SQL("select \* from user where id in (:1)")

public List<User> getUserList(@CacheBy List<Integer> ids);

@SQL("select id, name from user where id=:1 and name=:2")

public User getByUidAndName(@CacheBy int id, @CacheBy String name); //同时使用id和name两个字段来做缓存

}

**Mango事务**

mango支持编程式事务和使用TransactionTemplate进行事务；

**编程式事务：**

Transaction tx = TransactionFactory.newTransaction();

try {

//业务处理，比如下面的转账操作

dao.transferMoney(zhangsan, -money);

dao.transferMoney(lisi, money);

} catch (Throwable e) {

tx.rollback(); //事务回滚

return;

}

tx.commit(); //事务提交

**使用TransactionTemplate：**

TransactionTemplate.execute(new TransactionAction() {

@Override

public void doInTransaction(TransactionStatus status) {

//业务处理，比如下面的转账操作

dao.transferMoney(zhangsan, -money);

dao.transferMoney(lisi, money);

}

});

**Mango实时统计**

当我们使用mango框架访问数据库（或缓存）时，mango框架中的统计模块会实时的统计数据库（或缓存）的总访问次数，错误访问次数，平均响应时间，缓存命中率等数据。通过这些统计数据，我们能实时并全方位的了解我们的系统，方便我们更好的进行压力测试与线上实时性能监控。

**使用Web页面查看实时统计数据：**

最简单的查看实时统计的方法是使用web页面查看，如果我们在web项目中使用了mango框架，那么只需要将下面的配置文件添加到web项目的web.xml文件中，即可完成所有准备工作。

<servlet>

<servlet-name>mango-stat</servlet-name>

<servlet-class>org.jfaster.mango.plugin.stats.MangoStatServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>mango-stat</servlet-name>

<url-pattern>/mango-stat</url-pattern>

</servlet-mapping>

上面的配置文件，我们使用了一个类全名为 org.jfaster.mango.plugin.stats.MangoStatServlet 的servlet，并将它映射到 /mango-stat 路径下，假设web服务的ip为127.0.0.1，端口号为8080，那门我们只需要用浏览器访问htt://127.0.0.1:8080/mango-stat即可查看到实时统计数据。

**周期性生成实时统计数据：**

为了使统计数据能更准确的反映最近一段时间dao方法调用的性能，我们不能累积从服务开始运行到当前时间段所有的统计数据，而是应该将统计数据周期性的输出（比如以10秒为一个周期输出统计数据），为此mango框架提供了StatMonitor接口。

StatMonitor接口的类全名为org.jfaster.mango.stat.StatMonitor

public interface StatMonitor {

public int periodSecond();

public void handleStat(long statBeginTime, long statEndTime, List<OperatorStat> stats) throws Exception;

}

StatMonitor接口一共有两个需要实现的方法：

int periodSecond()，指定统计周期，单位为秒，比如当返回10时表示以10秒为一个统计周期。

void handleStat(long statBeginTime, long statEndTime, List<OperatorStat> stats)，每过一个统计周期会自动调用一次handleStat方法，所以我们可以将处理统计数据的逻辑放在handleStat方法中。参数statBeginTime为统计开始时间，参数statEndTime为统计结束时间，它们的单位都为毫秒；参数stats是一个OperatorStat对象的列表，对应前一个周期所有DAO方法的统计数据

## OperatorStat类统计数据表格

| **属性** | **类型** | **描述** |
| --- | --- | --- |
| method | java.lang.reflect.Method | SQL操作所在的方法 |
| operatorType | 枚举类型OperatorType | SQL操作类型 |
| isCacheable | boolean | 是否使用缓存 |
| initCount | long | SQL操作被初始化次数（一般情况为1） |
| databaseExecuteSuccessCount | long | 数据库执行成功数 |
| databaseExecuteExceptionCount | long | 数据库执行异常数 |
| databaseExecuteCount | long | 数据库执行总数（成功数＋异常数） |
| databaseExecuteSuccessRate | double | 数据库执行成功率 |
| databaseExecuteExceptionRate | double | 数据库执行异常率 |
| databaseExecuteAveragePenalty | long | 数据库执行平均响应时间（单位为纳秒） |
| hitCount | long | 缓存命中数 |
| missCount | long | 缓存丢失数 |
| hitRate | double | 缓存命中率 |
| cacheGetSuccessCount | long | 缓存get操作成功数 |
| cacheGetExceptionCount | long | 缓存get操作异常数 |
| cacheGetCount | long | 缓存get操作总数（成功数＋异常数） |
| cacheGetSuccessRate | double | 缓存get操作成功率 |
| cacheGetExceptionRate | double | 缓存get操作异常率 |
| cacheGetAveragePenalty | long | 缓存get操作平均响应时间（单位为纳秒） |
| cacheGetBulkSuccessCount | long | 缓存批量get操作成功数 |
| cacheGetBulkExceptionCount | long | 缓存批量get操作异常数 |
| cacheGetBulkCount | long | 缓存批量get操作总数（成功数＋异常数） |
| cacheGetBulkSuccessRate | double | 缓存批量get操作成功率 |
| cacheGetBulkExceptionRate | double | 缓存批量get操作异常率 |
| cacheGetBulkAveragePenalty | long | 缓存批量get操作平均响应时间（单位为纳秒） |
| cacheSetSuccessCount | long | 缓存set操作成功数 |
| cacheSetExceptionCount | long | 缓存set操作异常数 |
| cacheSetCount | long | 缓存set操作总数（成功数＋异常数） |
| cacheSetSuccessRate | double | 缓存set操作成功率 |
| cacheSetExceptionRate | double | 缓存set操作异常率 |
| cacheSetAveragePenalty | long | 缓存set操作平均响应时间（单位为纳秒） |
| cacheAddSuccessCount | long | 缓存add操作成功数 |
| cacheAddExceptionCount | long | 缓存add操作异常数 |
| cacheAddCount | long | 缓存add操作总数（成功数＋异常数） |
| cacheAddSuccessRate | double | 缓存add操作成功率 |
| cacheAddExceptionRate | double | 缓存add操作异常率 |
| cacheAddAveragePenalty | long | 缓存add操作平均响应时间（单位为纳秒） |
| cacheDeleteSuccessCount | long | 缓存delete操作成功数 |
| cacheDeleteExceptionCount | long | 缓存delete操作异常数 |
| cacheDeleteCount | long | 缓存delete操作总数（成功数＋异常数） |
| cacheDeleteSuccessRate | double | 缓存delete操作成功率 |
| cacheDeleteExceptionRate | double | 缓存delete操作异常率 |
| cacheDeleteAveragePenalty | long | 缓存delete操作平均响应时间（单位为纳秒） |
| cacheBatchDeleteSuccessCount | long | 缓存批量delete操作成功数 |
| cacheBatchDeleteExceptionCount | long | 缓存批量delete操作异常数 |
| cacheBatchDeleteCount | long | 缓存批量delete操作总数（成功数＋异常数） |
| cacheBatchDeleteSuccessRate | double | 缓存批量delete操作成功率 |
| cacheBatchDeleteExceptionRate | double | 缓存批量delete操作异常率 |
| cacheBatchDeleteAveragePenalty | long | 缓存批量delete操作平均响应时间（单位为纳秒） |

**Mango日志**

mango框架拥有强大的日志处理功能，能通过下面任意一个“日志工具”输出日志（能查看执行的SQL与参数）:

Slf4J

Log4J2

Log4J

Console(控制台，也就是通过System.out输出)

**使用Slf4J输出日志：**

准备配置文件

logback.xml 配置文件如下，需要注意的是logger的name为org.jfaster.mango，level为debug。

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} - %msg%n</pattern>

</encoder>

</appender>

<logger name="org.jfaster.mango" level="debug" additivity="false">

<appender-ref ref="STDOUT"/>

</logger>

<root level="error">

<appender-ref ref="STDOUT" />

</root>

</configuration>

Main方法运行程序：

有了前面的配置文件，当我们直接使用Main方法运行程序时，只需将下面的代码放到Main方法的最前面，即可让mango框架使用Slf4J输出日志信息。

MangoLogger.useSlf4JLogger();

上面的代码中MangoLogger的类全名为org.jfaster.mango.util.logging.MangoLogger

WEB容器运行程序：

很多时候我们并不直接通过Main方法运行自己的程序，而是使用tomcat，jetty等WEB容器。这时我们只需将下面的“监听器”拷配到web.xml文件的最前面，即可让mango框架使用Slf4J输出日志信息。

<listener>

<listener-class>org.jfaster.mango.plugin.listener.Slf4JLoggerListener</listener-class>

</listener>

**使用Log4J2输出日志：**

准备配置文件

log4j2.xml 配置文件如下，需要注意的是Logger的name为org.jfaster.mango，level为debug。

<?xml version="1.0" encoding="UTF-8"?>

<Configuration status="WARN">

<Appenders>

<Console name="Console" target="SYSTEM\_OUT">

<PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>

</Console>

</Appenders>

<Loggers>

<Logger name="org.jfaster.mango" level="debug" additivity="false">

<AppenderRef ref="Console"/>

</Logger>

<Root level="error">

<AppenderRef ref="Console"/>

</Root>

</Loggers>

</Configuration>

Main方法运行程序：

有了前面的配置文件，当我们直接使用Main方法运行程序时，只需将下面的代码放到Main方法的最前面，即可让mango框架使用Log4J2输出日志信息。

MangoLogger.useLog4J2Logger();

上面的代码中MangoLogger的类全名为org.jfaster.mango.util.logging.MangoLogger

WEB容器运行程序：

很多时候我们并不直接通过Main方法运行自己的程序，而是使用tomcat，jetty等WEB容器。这时我们只需将下面的“监听器”拷配到web.xml文件的最前面，即可让mango框架使用Log4J2输出日志信息。

<listener>

<listener-class>org.jfaster.mango.plugin.listener.Log4J2LoggerListener</listener-class>

</listener>

**使用Log4J输出日志：**

准备配置文件

log4j.xml 配置文件如下，需要注意的是logger的name为org.jfaster.mango，level为debug。

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration>

<appender name="console" class="org.apache.log4j.ConsoleAppender">

<layout class="org.apache.log4j.PatternLayout">

<param name="ConversionPattern" value="[%d{dd HH:mm:ss,SSS\} %-5p] [%t] %c{2\} - %m%n" />

</layout>

</appender>

<logger name="org.jfaster.mango" additivity="false">

<level value="debug" />

<appender-ref ref="console" />

</logger>

<root>

<level value="info" />

<appender-ref ref="console"/>

</root>

</log4j:configuration>

Main方法运行程序：

有了前面的配置文件，当我们直接使用Main方法运行程序时，只需将下面的代码放到Main方法的最前面，即可让mango框架使用Log4J输出日志信息。

**MangoLogger.useLog4JLogger();**

上面的代码中MangoLogger的类全名为org.jfaster.mango.util.logging.MangoLogger

WEB容器运行程序：

很多时候我们并不直接通过Main方法运行自己的程序，而是使用tomcat，jetty等WEB容器。这时我们只需将下面的“监听器”拷配到web.xml文件的最前面，即可让mango框架使用Log4J输出日志信息。

<listener>

<listener-class>org.jfaster.mango.plugin.listener.Log4JLoggerListener</listener-class>

</listener>

**使用控制台输出日志：**

使用System.out输出日志到控制台不需要配置文件。

Main方法运行程序：

只需要在Main方法的最前面加入如下代码，即可让Mango框架使用控制台输出日志信息：

**MangoLogger.useConsoleLogger();**

上面的代码中MangoLogger的类全名为org.jfaster.mango.util.logging.MangoLogger

WEB容器运行程序：

很多时候我们并不直接通过Main方法运行自己的程序，而是使用tomcat，jetty等WEB容器。这时我们只需将下面的“监听器”拷配到web.xml文件的最前面，即可让mango框架使用控制台输出日志信息。

**<listener>**

**<listener-class>org.jfaster.mango.plugin.listener.ConsoleLoggerListener</listener-class>**

**</listener>**

**Mango集成Spring**

在项目开发中，一般都会使用spring管理对象，进行依赖注入。 我们能通过mango自带的spring插件，便捷的将mango集成到spring中。

纯配置文件集成

纯配置文件集成是最简单的集成方式，所有的集成操作均在spring配置文件中，由spring容器创建数据库源工厂，mango对象和扫描使用@DB注解修饰的DAO类。

Mango框架使用SimpleDataSourceFactory连接单一数据库，

使用MasterSlaveDataSourceFactory连接主从数据库，

使用MultipleDatabaseDataSourceFactory连接混合数据库集群，

下面将分别给出这3种方式连数据库的配置实例。

连单一数据库配置实例:

<beans>

<!-- 配置数据源工厂 -->

<bean id="dsf" class="org.jfaster.mango.datasource.SimpleDataSourceFactory">

<property name="dataSource">

<bean class="org.jfaster.mango.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mango\_example" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

</property>

</bean>

<!-- 配置mango对象 -->

<bean id="mango" class="org.jfaster.mango.operator.Mango" factory-method="newInstance">

<property name="dataSourceFactory" ref="dsf" />

</bean>

<!-- 配置扫描使用@DB注解修饰的DAO类 -->

<bean class="org.jfaster.mango.plugin.spring.MangoDaoScanner">

<property name="packages">

<list>

<!-- 扫描包名 -->

<value>org.jfaster.mango.example.spring</value>

<!-- <value>其他需要扫描的包</value> -->

</list>

</property>

</bean>

</beans>

上面的配置主要包含3部分。

配置数据源工厂。DriverManagerDataSource数据源是mango框架对DataSource的简单实现，仅供测试使用，如果是生产环境，请使用HikariCP，c3p0，dbcp等高性能数据源。

配置mango对象。创建mango对象的最佳途径是通过Mango类的静态方法newInstance，所以使用了factory-method指定静态方法。

配置扫描使用@DB注解修饰的DAO类。MangoDaoScanner类是一个扫描DAO的扫描器，它能自动扫描packages属性中指定包下的所有类，识别出@DB注解修饰的DAO类，并将他自动加载到spring大工厂中，这样我们既能从ApplicationContext中直接getBean获得dao实例，也能将dao实例直接Autowired到所有由spring管理的类上。需要注意的是所有DAO类必须以DAO或Dao结尾，才能被扫描器识别。

连主从数据库配置实例:

<beans>

<!-- 配置主从数据源工厂 -->

<bean id="dsf" class="org.jfaster.mango.datasource.MasterSlaveDataSourceFactory">

<property name="master">

<bean class="org.jfaster.mango.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mango\_example\_master" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

</property>

<property name="slaves">

<list>

<bean class="org.jfaster.mango.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mango\_example\_slave1" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

<bean class="org.jfaster.mango.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mango\_example\_slave2" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

</list>

</property>

</bean>

<!-- 配置mango对象 -->

<bean id="mango" class="org.jfaster.mango.operator.Mango" factory-method="newInstance">

<property name="dataSourceFactory" ref="dsf" />

</bean>

<!-- 配置扫描使用@DB注解修饰的DAO类 -->

<bean class="org.jfaster.mango.plugin.spring.MangoDaoScanner">

<property name="packages">

<list>

<!-- 扫描包名 -->

<value>org.jfaster.mango.example.spring</value>

<!-- <value>其他需要扫描的包</value> -->

</list>

</property>

</bean>

</beans>

连多个数据库集群配置实例:

<beans>

<!-- 配置简单数据源工厂 -->

<bean id="simpleDataSourceFactory" class="org.jfaster.mango.datasource.SimpleDataSourceFactory">

<property name="name" value="dsf1" />

<property name="dataSource">

<bean class="org.jfaster.mango.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mango\_example" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

</property>

</bean>

<!-- 配置主从数据源工厂 -->

<bean id="masterSlaveDataSourceFactory" class="org.jfaster.mango.datasource.MasterSlaveDataSourceFactory">

<property name="name" value="dsf2" />

<property name="master">

<bean class="org.jfaster.mango.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mango\_example\_master" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

</property>

<property name="slaves">

<list>

<bean class="org.jfaster.mango.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mango\_example\_slave1" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

<bean class="org.jfaster.mango.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mango\_example\_slave2" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

</list>

</property>

</bean>

<!-- 配置mango对象 -->

<bean id="mango" class="org.jfaster.mango.operator.Mango" factory-method="newInstance">

<property name="dataSourceFactories">

<list>

<ref bean="simpleDataSourceFactory" />

<ref bean="masterSlaveDataSourceFactory" />

</list>

</property>

</bean>

<!-- 配置扫描使用@DB注解修饰的DAO类 -->

<bean class="org.jfaster.mango.plugin.spring.MangoDaoScanner">

<property name="packages">

<list>

<!-- 扫描包名 -->

<value>org.jfaster.mango.example.spring</value>

<!-- <value>其他需要扫描的包</value> -->

</list>

</property>

</bean>

</beans>

配置文件加代码集成:

一般情况下使用纯配置文件集成就能完成大多数项目需求，但在有些项目中，我们需要自己管理数据源工厂，以便在线上动态切换数据源。这时数据源工厂和mango对象对象就不能简单的交由spring管理，我们需要使用配置文件加代码的方式来完成集成。

下面是一个简单的配置文件加代码集成实例:

<beans>

<!-- 配置扫描使用@DB注解修饰的DAO类 -->

<bean class="org.jfaster.mango.plugin.spring.MangoDaoScanner">

<property name="packages">

<list>

<!-- 扫描包名 -->

<value>org.jfaster.mango.example.spring</value>

<!-- <value>其他需要扫描的包</value> -->

</list>

</property>

<property name="factoryBeanClass" value="org.jfaster.mango.example.spring.MyMangoFactoryBean" />

</bean>

</beans>

public class MyMangoFactoryBean extends AbstractMangoFactoryBean {

@Override

public Mango createMango() {

String driverClassName = "com.mysql.jdbc.Driver";

String url = "jdbc:mysql://localhost:3306/mango\_example";

String username = "root"; // 这里请使用您自己的用户名

String password = "root"; // 这里请使用您自己的密码

DataSource ds = new DriverManagerDataSource(driverClassName, url, username, password);

Mango mango = Mango.newInstance(ds); // 使用数据源初始化mango

return mango;

}

}

上面的实例分为spring配置文件与代码两部分。在spring配置文件中只有扫描DAO类的配置，并多了一个对扫描器MangoDaoScanner的factoryBeanClass属性的配置，factoryBeanClass的值是一个自定义的类MyMangoFactoryBean。代码部分，自定义类MyMangoFactoryBean继承了mango自带的抽象类org.jfaster.mango.plugin.spring.AbstractMangoFactoryBean，MyMangoFactoryBean通过实现createMango方法，实现用代码创建数据源工厂与mango对象。