Vastbase G100 V2.2

(Build 5)

应用适配手册

北京海量数据技术股份有限公司

目录

1. 适配一览表

适配组件	适配对象	适配版本	适配说明
连接池	Alibaba Druid	1.1.24	适配
ORM 框架	Mybatis	3.4.6	适配
	MybatisPlus	3.2.0	适配
	Hibernate	5.5.0 Final	适配
中间件	Tomcat	7.0.109	适配
	ShardingSphere-JDBC	4.1.1	适配
	ShardingSphere-Proxy	4.1.1	适配

2. 概述

概述

本章介绍如何数据库迁移至 Vastbase 后应用程序的兼容适配处理，以解决由于数据库不同特点导致原有应用运行出错的问题。

分为如下几个方面进行适配说明：

1. ORM 框架

2. 中间件

读者对象

本文档是为基于 Vastbase G100 进行 Java 应用程序开发的程序员而写的，提供了必要的参考信息。作为应用程序开发人员，至少需要了解以下知识：

- 操作系统知识。这是一切的基础。
- Java 语言。这是做应用程序开发的基础。
- 熟悉 Java 的一种 IDE。这是高效完成开发任务的必备条件。
- SQL 语法。这是操作数据库的必备能力。
- ORM 框架。这是适配这一部分的基础
- 中间件 Tomcat 和 ShardingSphere。同样是适配这一部分的必要条件

3. ORM 框架适配

3.1. 驱动及连接池配置

我们知道对于不同的数据库我们需要配置不同的数据库驱动，在 Springboot 项目中，我们可以在 application.yml 中进行配置。如对于 MySQL8.0 版本，需要进行如下操作：

Step1:在 pom.xml 中导入依赖：

```
<!--          数据库-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.12</version>
        </dependency>
```

Step2:在 application.yml 中进行配置

```
#配置数据库链接
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root
    password:
    url: jdbc:mysql://127.0.0.1:3306/expressproject?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai&useSSL
```

驱动四要素即：driver、username、password 以及 url，这里使用的数据库连接池为默认连接池。

而对于 vastbase，则需要导入 vastbase 的驱动，以 2.0 版本为例：

Step1:Maven 本地导入 Vastbase-G100-2.0_xxxxxxxxxx.jar 依赖

```
mvn install:install-file "-Dfile=Vastbase-G100-x.x_xxxxxxx.jar" "-DgroupId=cn.com.vastdata"
"-DartifactId=vastbase" "-Dversion=2.0" "-Dpackaging=jar"
```

Step2:在 pom.xml 中导入

```
<dependency>
    <groupId>cn.com.vastdata</groupId>
    <artifactId>vastbase</artifactId>
    <version>2.0</version>
</dependency>
```

Step3:配置数据源和驱动

```yaml
#配置
spring:
  datasource:
    #配置数据源类型
    type:
      com.alibaba.druid.pool.DruidDataSource
    driver-class-name: org.postgresql.Driver
    username
    password:
    url: jdbc:postgresql
    #初始化，最小，最大连接数
    initialSize: 3
    minidle: 3
    maxActive: 10
    #获取数据库连接等待的超时时间
    maxWait: 60000
    #配置多久进行一次检测，检测需要关闭的空闲连接
    timeBetweenEvictionRunsMillis: 60000
    validationQuery: SELECT 1 FROM dual
    #配置监控统计拦截的filters,去掉后，健康界面的sql无法统计
    filters: stat,wall,log4j
```

同样只需要配置驱动四要素：driver、username、password 以及 url

而其中的 com.alibaba.druid.pool.DruidDataSource 是市面上较为流行的性能也相对较高的数据库连接池 druid，除此之外还有 C3P0,DBCP 等主流数据库连接池。不同连接池共同的地方在于需要配置驱动四要素，尽管命名可能有略微不同。我们只需要根据其官方文档来进行命名配置即可。

而如上所示的 druid 连接池中我们还可以配置多个属性如 initialSize 等，可以根据项目的需要进行相关配置。

3.2. Mybatis

3.2.1. 类名调整

Mybatis 需要用到类名时，需要写入类的全路径，这样无疑耗费时间，如下例当我们需要使用 cn.com.atlasdata.test.mapper 包下的 TestBooleanMapper 类时，在 Mapper 中的写法如下：

```xml
<insert id="insert" parameterType="cn.com.atlasdata.test.mapper.TestBooleanMapper">
    insert into hcj.test_boolean(column1,id) values(#{column1},#{id});
</insert>
```

也就是当我们的 resultType 或 parameterType 为实体类时，我们需要写这个类的全路径，否则将会报错如下：

```
- Enabling autowire by type for MapperFactoryBean with name 'testBooleanMapper'.
```

Mybatis 提供了一种简便的方法解决这一问题，也就是“扫描包变小写”

```yaml
#mybatis的配置
mybatis:
  type-aliases-package: cn.com.atlasdata.test.pojo
```

我们只需要在 application.yml 中配置这一段，在 Mapper.xml 中便可以使用类名小写来替换类名的全路径，如下：

```
<insert id="insert" parameterType="testboolean">
    insert into hcj.test_boolean(column1,id) values(#{column1},#{id});
</insert>
```

但是，在不断的尝试过程中，我发现这一方法并没有针对性得要求一定是小写，而是不区分大小写，也就是说这这里我们使用 Testboolean、TESTBoolean 都是可以的。

3.2.2. 字段名称转换

由于 Windows 环境下数据库不区分大小写，因此我们对表名或字段名进行命名时通常使用下划线如 user_id。但是，我们在 Java 开发中，为了遵守开发规范，命名一般为驼峰命名，如 userId。这样,Mybatis 就可能报错没有 user_id 的 get 和 set 方法。

如下例：

数据库的 user 表

字段名	#	数据类型	长度	精度	标度	Identity	Collation	非空	默认值	描述
123 user_id	1	int4		10				☐	nextval('hcj.u...	用户Id
ABC username	2	varchar					default	☐		用户名
ABC pwd	3	varchar					default	☐		密码
ABC role	4	varchar					default	☐		角色
ABC phone	6	varchar					default	☐		用户手机号

Java 的 User 类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private int userId;
    private String pwd;
    private String username;
    private String role;
    private String phone;
}
```

```
<insert id="addUser" parameterType="user">
    insert into hcj.user values(#{userId},#{username},#{pwd},#{role},#{phone});
</insert>
```

此时，我们搭建好 Mybatis 并运行，便会出现如下情况：

```
[19:45:29:115] [INFO] - org.postgresql.log.JdkLogger.logInner(JdkLogger.java:135) - Connect complete. ID: ae559baa-6b6a-4f78-b963-1f4bc12eb21a
[19:45:29:152] [DEBUG] - org.mybatis.spring.transaction.SpringManagedTransaction.openConnection(SpringManagedTransaction.java:87) - JDBC Connection [org.postgresql.jdbc.PgConnection@62ee5d2] will not be managed by Spring
[19:45:29:158] [DEBUG] - org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java:159) - ==>  Preparing: insert into hcj.user values(?,?,?,?,?);
[19:45:29:180] [DEBUG] - org.mybatis.spring.SqlSessionUtils.closeSqlSession(SqlSessionUtils.java:191) - Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@24dbb048]
[19:45:29:190] [ERROR] - org.apache.juli.logging.DirectJDKLog.log(DirectJDKLog.java:175) - Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception
org.apache.ibatis.reflection.ReflectionException Create breakpoint : There is no getter for property named 'user_id' in 'class cn.com.atlasdata.test.pojo.User'
    at org.apache.ibatis.reflection.Reflector.getGetInvoker(Reflector.java:419) ~[mybatis-3.4.6.jar:3.4.6]
    at org.apache.ibatis.reflection.MetaClass.getGetInvoker(MetaClass.java:184) ~[mybatis-3.4.6.jar:3.4.6]
    at org.apache.ibatis.reflection.wrapper.BeanWrapper.getBeanProperty(BeanWrapper.java:160) ~[mybatis-3.4.6.jar:3.4.6]
    at org.apache.ibatis.reflection.wrapper.BeanWrapper.get(BeanWrapper.java:69) ~[mybatis-3.4.6.jar:3.4.6]
```

因此，我们需要将数据库的下划线命名和 Java 开发的驼峰命名进行转换。解决这一问题有多种方法，但最简便的一种便是 Mybatis 提供的下划线驼峰命名转换。

（1）Mybatis 提供了十分简便的解决方法，我们只需要在 yml 的配置中加入这一段，也就是配置下划线转驼峰，而后 Mybatis 便会自动将 SQL 中查出来的带下划线的字段转换为驼峰命名格式，再去匹配类中的属性，问题便得已解决。

```
mybatis:
  configuration:
    map-underscore-to-camel-case: true
```

（2）也可以通过给字段起别名的方式来解决

```
select user_id as userId
from hcj.user
```

（3）用 resultMap 映射

```
<select id="getUserList" resultMap="selectUser">
    select * from hcj.user;
</select>
<resultMap id="selectUser" type="User">
    <id column="user_id" property="id"/>
    <result column="pwd" property="pwd"/>
    <result column="username" property="username"/>
    <result column="role" property="role"/>
    <result column="phone" property="phone"/>
</resultMap>
```

3.2.3. 字段值处理

3.2.3.1. Null 值处理

一般而言，对于数据库中的字段，其值可能为 null，因此在 Java 类中的属性类型设置时，我们应当使用包装类即 Integer 代替 int 等，使其可与数据库字段取值范围对应。

3.2.3.2. 其他数据类型处理

在 Vastbase 中，采用的是 Postgresql 的语法，因此同样也支持 Postgresql 的相关基本类型，这些数据类型与 Java 数据类型的对应关系如下

（1）数值类型

数据库基本类型	Java 类型
tinyint	Integer
smallint	Integer
integer	Integer
bigint	Long
numeric	BigDecimal
real	Float
smallserial	Integer

serial	Integer
bigserial	Long
oid	Long

（2）货币类型

数据库基本类型	Java 类型
money	BigDecimal

注意货币类型 Money 对应的是 Java 类型中的 BigDecimal 而不是 Float 或者 Double，原因在于 Float 和 Double 都是近似值，当用于金额的加减乘除时，可能会出错，这在金额的计算中显然是不被允许的，使用 BigDecimal 便能很好解决这一点。

（3）字符类型

数据库基本类型	Java 类型
char	Charater 或 String(前端传参需要注意字符数只能为 1)
bpchar	Charater 或 String(前端传参需要注意字符数只能为 1)
varchar	String
nvarchar2	String
text	String
name	String

（4）时间类型

数据库基本类型	Java 类型
timestamp	Timestamp
timestampz	Timestamp
date	Date
time	Time
timez	Time
smalldatetime	Timestamp

注：在 Mybatis 操作数据库的过程中，时间类型较为特殊，如下例：

数据库中的字段定义如下:

column1	1	timestamp		29	6		☐	timestamp类...
column2	2	timestamptz		35	6		☐	timpstamptz...
column3	3	timestamp		22			☐	date类型兼容...
column4	4	time		15	6		☐	time类型兼容...
column5	5	timetz		21	6		☐	timez类型兼...

Java 类中的属性定义如下:

```
@JsonFormat(shape = JsonFormat.Shape.STRING,pattern = "yyyy-MM-dd HH:mm:ss")
private Timestamp column1;
@JsonFormat(shape = JsonFormat.Shape.STRING,pattern = "yyyy-MM-dd HH:mm:ss")
private Timestamp column2;
@JsonFormat(shape = JsonFormat.Shape.STRING,pattern = "yyyy-MM-dd HH:mm:ss")
private Date column3;
@JsonFormat(shape = JsonFormat.Shape.STRING,pattern = "HH:mm:ss")
private Time column4;
@JsonFormat(shape = JsonFormat.Shape.STRING,pattern = "HH:mm:ss")
private Time column5;
```

需要在 Java 属性中加入如图所示的@JsonFormat 的注解

（5）位类型

数据库基本类型	Java 类型
bit	Integer
bytea	byte[]

这里同样需要注意的是,百度上以及各资料都显示 bit 对应的 Java 类型应当是 boolean，理由在于：boolean 的 true 和 false 映射到数据库时会变化为 0，1。但经过实际验证，发现此做法并不可取，无法适配。因此使用 Integer,用户在前端传递参数时将参数设置为只能是 0 或 1 即可。

而 bytea 对应的便是 byte[]，也就是 byte 数组。

且这两个类型存在特殊之处,由于在 Postgresql 语法中，对这两个类型的数据插入操作如下：

```
insert into hcj.test_char(id,column7,column8 ) values(1,'bye'::bytea,1::bit);
```

也就是强制类型转换。因此，在 Mybatis 增删改查时，同样需要进行一定处理，以插入数据操作为例：

```
<insert id="add" parameterType="testchar">
    insert into hcj.test_char(column1,column2,column3,column4,column6,column7,column8,column9,id)
    values(#{column1},#{column2},#{column3},#{column4},#{column6},#{column7}::bytea,#{column8}::bit,#{column9},#{id})
</insert>
```

（6）其它类型

数据库基本类型	Java 类型
json	Object
point	Object

这两个类型可以直接用 Object 类型进行处理。

在传入参数时，point 类型注意传入的应为：（1.0，2.0)这样的“点”；

而 json 数据与 bit 以及 bytea 同理，需要进行特殊处理

```
,#{column8}::json)
```

3.2.4. 结果集封装

3.2.4.1. Map 封装

Mybatis 在将结果集封装成 resultType=“map”的时候，key 值为小写，客户应用中在处理 map 中的数据时，都默认为大写。

解决方案：

Step1：自定义 Map 包装器 MapKeyUpperWrapper 继承 MapWrapper,重写 findProperty()方法:

```
public class MapKeyUpperWrapper extends MapWrapper {
    public MapKeyUpperWrapper(MetaObject metaObject, Map<String, Object> map) {
        super(metaObject, map);
    }

    @Override
    public String findProperty(String name, boolean useCamelCaseMapping) {
        return name==null?"":name.toUpperCase() ;
    }

}
```

Step2：自定义 Map 包装器工厂类 MapWrapperFactory，实现 ObjectWrapperFactory 接口，并实现接口方法：

```
public class MapWrapperFactory implements ObjectWrapperFactory {
    @Override
    public boolean hasWrapperFor(Object object) {
        return object != null && object instanceof Map;
    }

    @Override
    public ObjectWrapper getWrapperFor(MetaObject metaObject, Object object) {
        return new MapKeyUpperWrapper(metaObject, (Map) object);
    }
}
```

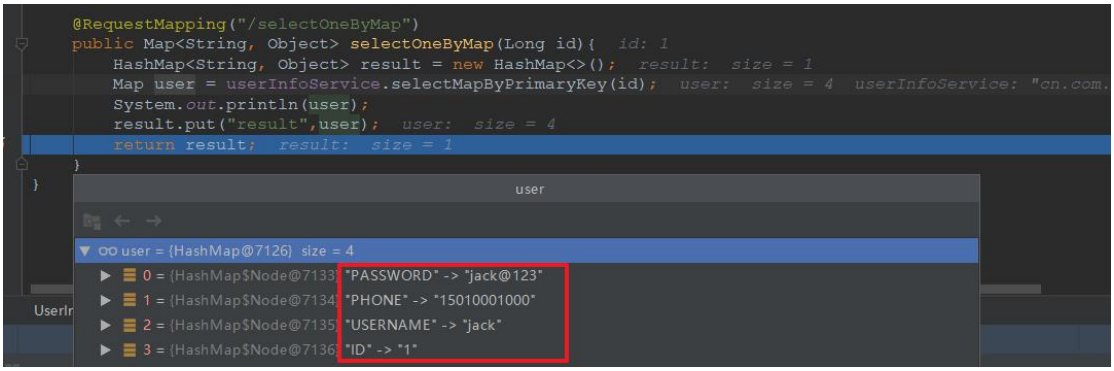
Step3：由于 Mybatis 与 SpringBoot 不提供用反射机制来构建对象的 converter，但是 SpringBoot 提供了允许自定义的 converter 来进行转换，所以这里需要构建一个自定义的 converter：

```
@Component
@ConfigurationPropertiesBinding
public class ObjectFactoryConverter implements Converter<String, ObjectWrapperFactory> {

    @Override
    public ObjectWrapperFactory convert(String source) {
        try {
            return (ObjectWrapperFactory) Class.forName(source).newInstance();
        } catch (InstantiationException | IllegalAccessException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Step4：最后一步，在配置文件中，对自定义的 MapWrapperFactory 进行配置：

```
#配置自定义的对象包装器
mybatis.configuration.object-wrapper-factory=cn.com.atlasdata.ibdemo.base.util.MapWrapperFactory
```



3.2.5. 动态 SQL

在实际应用中，可能存在传进来的参数为一个或者多个的情况，这时候动态 sql 是一个很好的帮手，通过动态 sql，Mybatis 便可以根据对应的参数进行相关条件的数据查询等操作。如下简单例子

```
<update id="updateUserById" parameterType="int">
    update hcj.user
    <set>
        <if test="username ! =null"> username=#{username},</if>
        <if test="pwd !=null"> pwd=#{pwd},</if>
        <if test="role !=null"> role=#{role},</if>
        <if test="phone !=null"> phone=#{phone}</if>
    </set>
</update>
```

3.2.6. 存储过程及函数

3.2.6.1. Vastbase 语法方面

（1）带参数带返回的函数

test_int 表：

函数：

返回类型为表时：

```
create or replace function selectall(id1 int) returns setof hcj.test_int as
$$
begin
return query select * from hcj.test_int where id=id1;
end;
$$language plpgsql;
```

返回类型为字段时：

```
create or replace function selectid(id1 int) returns setof int as
$$
begin
return query select column2 from hcj.test_int where id =id1;
end;
$$language plpgsql;
```

其中 test_int 为表名，当我们的函数带有返回类型时，需要使用“returns setof+类型”的形式表明返回类型，且返回语句的语法为 return query...，如上图所示。

（2）带参数无返回的函数

```
create or replace function deleteid(id1 int) returns void as
$$
begin
delete from hcj.test_int where id=id1;
end;
$$language plpgsql;
```

无返回时，与 Java 语法类似的，我们需要 returns void

（3）不带参数带返回的函数

```
create or replace function addid() returns setof int as
$$
begin
insert into hcj.test_int
values('400','400','400','400','400','400','400','400','400','7');
return next 1;
end;
$$language plpgsql;
```

return next 1；即返回一个 int 类型的数据 1

（4）不带参数无返回的函数

```
create or replace function updateid() returns void as
$$
begin
update hcj.test_int
set column1=700
where id=3;
end;
$$language plpgsql;
```

带参数时，即在函数名后加上“参数名 参数类型”,带返回值时则需要注意 returns setof +返回类型以及 return query 返回语句。不带返回类型时需注意 returns void

3.2.6.2. 函数与 Mybatis 结合使用

首先在 TestIntMapper 接口中定义相关方法

```
void testFunctionUpdate();
TestInt testFunctionSelectAll(Integer id);
Integer testFunctionSelectId(Integer id);
void testFunctionDeleteId(Integer id);
Integer testFunctionAddId();
```

而后使用即可

```
<select id="testFunctionSelectAll" resultType="testint">
    <![CDATA[
    select *
    from public.selectall(#{id});
    ]]>
```

注：这里的<![CDATA[ ]]>代表执行函数或存储过程

3.3. MybatisPlus

3.3.1. MybatisPlus 的数据库连接问题

问题产生：在使用代码自动生成器时，报错找不到表格，但实际在数据库中该表格存在

解决方法：在数据库连接的配置中加上 currentSchema 的配置，如下图，否则在使用代码自动生成器的过程中会默认是找 public 下的表，自然就会报错。

```yaml
#配置
spring:
  datasource:
    #配置数据源类型
    type:
      com.alibaba.druid.pool.DruidDataSource
    driver-class-name: org.postgresql.Driver
    username: vbadmin
    password: Vbase@admin
    url: jdbc:postgresql://172.16.103.120:7032/vastbase?currentSchema=hcj
    #初始化，最小，最大连接数
    initialSize: 3
    minidle: 3
    maxActive: 10
    #获取数据库连接等待的超时时间
    maxWait: 60000
    #配置多久进行一次检测，检测需要关闭的空闲连接
    timeBetweenEvictionRunsMillis: 60000
    validationQuery: SELECT 1 FROM dual
    #配置监控统计拦截的filters,去掉后，健康界面的sql无法统计
    filters: stat,wall,log4j
```

3.3.2. 设置数据库表中的创建时间字段和更新时间字段的自动插入

（1）数据库表的字段

create_time	8	timestamp
update_time	9	timestamp

（2）在字段上使用注解

```java
    @ApiModelProperty(value = "创建时间")
    @TableField(fill = FieldFill.INSERT)
    private LocalDateTime createTime;

    @ApiModelProperty(value = "更新时间")
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private LocalDateTime updateTime;
```

（3）配置自动插入和更新

```java
import java.time.LocalDateTime;
import java.util.Date;

/**
 * 处理自动生成时间
 */
@Configuration
public class MyObjectHandler implements MetaObjectHandler {
    @Override
    public void insertFill(MetaObject metaObject) {
        setFieldValByName( fieldName: "createTime", LocalDateTime.now(),metaObject);
        setFieldValByName( fieldName: "updateTime",LocalDateTime.now(),metaObject);
    }

    @Override
    public void updateFill(MetaObject metaObject) {
        setFieldValByName( fieldName: "updateTime",LocalDateTime.now(),metaObject);
    }
}
```

（4）配置完毕后在对表进行数据的插入或更新时，则会对这两个字段进行进行时间的插入或更新

	id	username	pwd	role	phone	deleted	version	create_time	update_time
1	1	admin	111117	快递员	19927539545	[NULL]		2021-06-17 15:31:13	2021-06-17 15:46:58
2	2	root	123456	快递员	19927539545	[NULL]	[NULL]	2021-06-21 18:12:20	2021-06-21 18:12:20

3.3.3. 乐观锁和逻辑删除字段的配置

3.3.3.1. 逻辑删除字段

（1）设计思想

即在数据库表中新设置一个字段，这里为 deleted。这一字段的设置思想为：当 deleted=1 时，该条记录对于用户不可见，但其在数据库中依然存在。本质上就是在查询的 SQL 语句中加上 and deleted=0; 如下：

执行 delete

```
==>  Preparing: UPDATE my_user SET deleted=1 WHERE id=? AND deleted=0
==> Parameters: 1(Integer)
<==    Updates: 1
```

查看数据库表，此时数据还在，但是 deleted 字段变为 1

	id	username	pwd	role	phone	deleted	version	create_time	update_time
1	2	root	123456	快递员	19927539545	0	0	2021-06-21 18:12:20	2021-06-21 18:12:20
2	1	admin	111117	快递员	19927539545	1	2	2021-06-17 15:31:13	2021-06-17 15:46:58

执行查询操作，可以看到 SQL 后面加上了 and deleted=0，查询的结果为空

```
SELECT id,deleted,role,create_time,phone,update_time,pwd,version,username FROM my_user WHERE id=? AND deleted=0
1(Integer)
0
```

（2）逻辑删除字段的配置

首先在字段上添加注解

```
    @TableLogic
    private Integer deleted;
```

到 application.yml 中配置 mybatis-plus 的逻辑删除字段

```
#mybatis-plus的配置
mybatis-plus:
  type-aliases-package: mybatisPlus.pojo
  mapper-locations: classpath*:mybatisPlus/mapper/xml/*.xml
  global-config:
    db-config:
      logic-delete-field: deleted  # 全局逻辑删除的实体字段名(since 3.3.0,配置后可以忽略不配置步骤2)
      logic-delete-value: 1 # 逻辑已删除值(默认为 1)
      logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)
```

3.3.3.2. 乐观锁字段

乐观锁的设计思路为：新建一个字段 version，取出记录时，获取当前的 version,更新时带上这个 vesion，执行

set version=newVersion where version=oldVersion

配置：

```
    @ApiModelProperty(value = "乐观锁字段")
    @Version
    private Integer version;
```

```java
package mybatisPlus.config;

import com.baomidou.mybatisplus.annotation.DbType;
import com.baomidou.mybatisplus.extension.plugins.OptimisticLockerInterceptor;
import com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor;
import com.baomidou.mybatisplus.extension.plugins.pagination.optimize.JsqlParserCountOptimize
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
@MapperScan("mybatisPlus.mapper.xml.MyUserMapper.xml")
public class MybatisPlusConfig{
        /**
         *
         */
        @Bean
        public OptimisticLockerInterceptor optimisticLockerInterceptor() {
            return new OptimisticLockerInterceptor();
        }
```

注意在乐观锁使用时，需要先查询出数据，而后更新时带上这个 version

```java
    @PostMapping("/update")
    private void update(@RequestBody UserForm userForm){
        MyUser myUser1 = myUserMapper.selectById(userForm.getId());
//
//        UpdateWrapper<MyUser> userUpdateWrapper= new UpdateWrapper<>();
//        userUpdateWrapper.eq("id",userForm.getId())
//                .set("phone",userForm.getPhone())
//                .set("role",userForm.getRole())
//                .set("pwd",userForm.getPwd())
//                .set("username",userForm.getUsername());
        MyUser myUser = new MyUser();
        myUser.setPwd(userForm.getPwd());
        myUser.setVersion(myUser1.getVersion());
        myUser.setId(userForm.getId());
        myUserMapper.updateById(myUser);
    }
```

如下例：

Step1:先查询

```
 Preparing: SELECT id,deleted,role,create_time,phone,update_time,pwd,version,username FROM my_user WHERE id=? AND deleted=0
Parameters: 2(Integer)
   Columns: id, deleted, role, create_time, phone, update_time, pwd, version, username
       Row: 2, 0, 快递员, 2021-06-21 18:12:20.776, 19927539545, 2021-06-21 18:12:20.776, 123456, 0, root
     Total: 1
```

Step2:后修改

```
==>  Preparing: UPDATE my_user SET update_time=?, pwd=?, version=? WHERE id=? AND version=? AND deleted=0
==> Parameters: 2021-06-21T18:32:07.012(LocalDateTime), 111117(String), 1(Integer), 2(Integer), 0(Integer)
<==    Updates: 1
```

Step3:查看数据库，可以观察到 version 值更新

2	root	111117	快递员	19927539545	0	1	2021-06-21 18:12:20	2021-06-21 18:32:07

3.3.4. 分页配置（MybatisPlus 分页插件的使用）

3.3.4.1. 使用分页的原因

数据库表中的数据可能非常巨大，如果不进行分页处理，一次性查询全部数据可能导致许多问题如查询效率低等，MybatisPlus 提供了分页插件

3.3.4.2. 分页插件的本质

分页插件的本质为在 SQL 语句后加上 limit ? offet ?

3.3.4.3. 分页插件的配置和使用

Step1:按照官网配置即可

```
@Bean
public PaginationInterceptor paginationInterceptor() {
    PaginationInterceptor paginationInterceptor = new PaginationInterceptor();
    return paginationInterceptor;
}
```

Step2:而后使用 MybatisPlus 自身封装的方法即可

```
@Test
void test_charSelectMapsPage(){
    IPage<TestChar> intIPage=new Page<>( current: 2, size: 4);
    QueryWrapper<TestChar> wrapper = new QueryWrapper<>();
    wrapper.eq( column: "column1", val: "8");
    System.out.println(testCharMapper.selectMapsPage(intIPage,wrapper));
}
```

Step3:测试，可以看到本质便是加上限制条件

```
==> Parameters: 2(String)
<==    Columns: count
<==        Row: 6
==>  Preparing: SELECT id,column1,column10,column5,column4,column3,column2,column9,column8,column7,column6 FROM test_char WHERE (column1 = ?) limit ? offset ?
==> Parameters: 2(String), 2(Long), 2(Long)
<==    Columns: id, column1, column10, column5, column4, column3, column2, column9, column8, column7, column6
<==        Row: 14, 2, null, 14, 14, 14, 14, $1.00, null, t, <<BLOB>>
<==        Row: 8, 2, null, 1, 1, 1, 1, $1.00, null, t, <<BLOB>>
<==      Total: 2
lSession [org.apache.ibatis.session.defaults.DefaultSqlSession@309dcdf3]
```

3.3.5. 特殊类型的字段后端传递参数

3.3.5.1. 几何类型 point

数据库中字段类型为 point，Java 中使用 Object 作为其类型，在后端传参时，使用 PGpoint 进行传参。

数据库中的 point

123 id	1	int4		10			☑	nex
123 column1	2	int2		5			☐	
123 column2	3	int4		10			☐	
123 column3	4	int8		19			☐	
123 column4	5	numeric					☐	
123 column5	6	float4		8	8		☐	
123 column6	7	float8		17	17		☐	
123 column7	8	int1					☐	
123 column8	9	oid		10			☐	
column9	10	point					☐	

Java 中使用 Object

```
@ApiModelProperty(value = "point")
private Object column9;
```

后端传递，使用 PGpoint

```
PGpoint pGpoint=new PGpoint();

/**
 * test_int
 **/
@Test
void test_intAdd() {
    for(int i=20;i<30;i++) {
        pGpoint.setLocation(i,i);
        testIntService.save(new TestInt(i, column1: 20, i, new Long(i), new BigDecimal(i),
                new Float(i), new Double(i), i, new Long(i), pGpoint));
    }
}
```

3.3.5.2. Json 类型、bit 类型以及 bytea 类型

这三个类型都无法使用 MybatisPlus 封装的方法进行插入操作，因此需要我们自定义插入的方法，编写 SQL 语句，如下：

数据库表 test_char 里面的字段 json、bytea、bit

字段名	#	数据类型	长度	精度	标度	Identity	Collation	非空	默认值	描述
id	1	int4		10				☑	nextval('hcj.te...	id
column1	2	bpchar	1	1			default	☐		
column2	3	bpchar					default	☐		bpchar
column3	4	varchar					default	☐		varchar
column4	5	text					default	☐		text
column5	6	name						☐		name
column6	7	bytea						☐		bytea
column7	8	bool	1	1				☐		bool
column8	9	bit		1				☐		bit
column9	10	money						☐		money
column10	11	json						☐		json

在 Mapper 中定义函数

```
public interface TestCharMapper extends BaseMapper<TestChar> {
    void addIntoChar(TestChar testChar);
}
```

Mapper.xml 实现

```
<insert id="addIntoChar" parameterType="testchar">
    insert into hcj.test_char
    values(#{id},#{column1},#{column2},#{column3},#{column4},#{column5},
    #{column6}::bytea,#{column7},#{column8}::bit,#{column9},#{column10}::json);
</insert>
```

即类型转换，也可通过调用函数的方式实现上述操作

其它类型与 Mybatis 的操作一致

3.3.6. MybatisPlus 封装的方法

3.3.6.1. Mapper crud 接口

在 Vastbase 数据库中，各种基本数据类型对于 Mapper 的各种 crud 接口都是适配的。其中，一些特殊字段的处理如 3.3.5 所示。

3.3.6.2. Service crud 接口

在 Vasebase 数据库中，各种基本数据类型对于 Service 的各种 crud 接口都是适配的。其中，一些特殊字段的处理如 3.3.5 所示。

3.3.7. 函数和存储过程

MybatisPlus 封装的方法并无法解决函数和存储过程的问题，因此对于函数和存储过程，操作与 Mybatis 一致

3.4. Hibernate

3.4.1. Hibernate 配置数据库的连接

（1）配置数据库连接，这部分与 Mybatis 和 MybatisPlus 一致，同样也需要 Vastbase 的依赖

```yaml
#配置
spring:
  datasource:
    #配置数据源类型
    type:
      com.alibaba.druid.pool.DruidDataSource
    driver-class-name: org.postgresql.Driver
    username: vbadmin
    password: Vbase@admin
    url: jdbc:postgresql://172.16.103.120:7032/vastbase?currentSchema=hcj
    #初始化，最小，最大连接数
    initialSize: 3
    minidle: 3
    maxActive: 10
    #获取数据库连接等待的超时时间
    maxWait: 60000
    #配置多久进行一次检测，检测需要关闭的空闲连接
    timeBetweenEvictionRunsMillis: 60000
    validationQuery: SELECT 1 FROM dual
    #配置监控统计拦截的filters,去掉后，健康界面的sql无法统计
    filters: stat,wall,log4j
```

（2）配置 jpa

```yaml
jpa:
  show_sql: true
  database-platform: org.hibernate.dialect.PostgreSQL9Dialect
  # database-platform: cn.luutqf.springboot.dialect.JsonbPostgresDialect
  hibernate:
    ddl-auto: update  # none: 关闭hibernate的自动创建表结构的机制
  properties:
    hibernate:
      dialect: org.hibernate.dialect.PostgreSQLDialect
      hbm2ddl.auto: update
      jdbc.lob.non_contextual_creation: true
      format_sql: true
      temp:
        # 兼容SpringBoot2.X, 关闭 Hibernate尝试验证PostgreSQL的CLOB特性
        use_jdbc_metadata_defaults: false
```

3.4.2. 实体类需要注意的事项

```java
@Data
@EqualsAndHashCode(callSuper = false)
@Accessors(chain = true)
@AllArgsConstructor
@NoArgsConstructor
@Entity(name = "testInt")
public class TestChar implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private Integer id;
```

注意点 1：实体类上需要有注解@Entity

注意点 2：id 字段需要有注解@Id

3.4.3. 持久层 Repository 需要注意的事项

```java
@Table(name = "testInt")
@Repository("TestIntRepository")
public interface TestIntRepository extends JpaRepository<TestInt,String> {
    @Query(value = "select * from test_int d where d.id =?1",nativeQuery = true)
    TestInt find(@Param("id") int id);//查询全部

    @Modifying
    @Transactional
    @Query(value = "update test_int d set d.column1=?2,d.column2=?3,d.column3=?4,d.column4=?5,d.column5=?6,d.column6=?7,d.column7=?8,d.column8=?9 where d.id=?1",nativeQuery = true)
    int myUpdate(int id, Integer column1, Integer column2, Long column3, BigDecimal column4, Float column5, Double column6,Integer column7,Long column8);

    @Modifying
    @Transactional
    @Query(value = "delete from test_int d where d.id = :id",nativeQuery = true)
    int myDelete(int id);
}
```

注意点 1：类上需要有对应实体类注解如@Table(name=“testInt”)

注意点 2：持久层注解标志@Repository(“TestIntRepository”)

注意点 3：持久类继承 JpaRepository<Object,String>，将 Object 更改为对应实体类

注意点 4：Hibernate 执行默认是 HQL 语句，而使用 nativeQuery=true 则将其转变为 SQL 语句语法

注意点 5：更新删除等操作时需要加上注解@Transactional，执行事务操作

注意点 6：@Modifying+@Query 定义个性化更新操作

3.4.4. 各种基本类型的操作

3.4.4.1. 时间类型

字段名	#	数据类型	长度	精度	标度	Identity	Collation	非空	默认值	描述
id	1	int4		10				☑	nextval('h...	id
column1	2	timestamp		22				☐		date
column2	3	time		15	6			☐		time
column3	4	timetz		21	6			☐		timetz
column4	5	timestamp		29	6			☐		timestamp
column5	6	smalldatetim						☐		smalldate...
column6	7	timestamptz		35	6			☐		timestampz

```
    private static final long serialVersionUID = 1L;
    @Id
    private Integer id;

    private Date column1;

    private Time column2;

    private Time column3;

    private LocalDateTime column4;

    private LocalDateTime column5;

    private LocalDateTime column6;
}
```

```
@Test
void find() { System.out.println(testDateRepository.find( id: 3)); }

@Test
void save(){
    Time time=new Time(1000011);
    testDateRepository.save(new TestDate( id: 2,new Date(),time,time, LocalDateTime.now(),LocalDateTime.now(),LocalDat
}

@Test
void update(){
    Time time=new Time(1000011);
    testDateRepository.myUpdate( id: 2,new Date(),time,time, LocalDateTime.now(),LocalDateTime.now(),LocalDateTime.now
}

@Test
void delete() { testDateRepository.myDelete( id: 1); }
```

3.4.4.2. 整型

字段名	#	数据类型	长度	精度	标度	Identity	Collation	非空	默认值	描述
id	1	int4		10				☑	nextval('h...	id
column1	2	int2		5				☐		smallint
column2	3	int4		10				☐		int
column3	4	int8		19				☐		bigint
column4	5	numeric						☐		numeric
column5	6	float4		8	8			☐		real
column6	7	float8		17	17			☐		double pr...
column7	8	int1						☐		tinynit
column8	9	oid		10				☐		oid
column9	10	point						☐		point
column10	13	varchar	255	255			default	☐		

```java
    private static final long serialVersionUID = 1L;

    @Id
    private Integer id;

    private Integer column1;

    private Integer column2;

    private Long column3;

    private BigDecimal column4;

    private Float column5;

    private Double column6;

    private Integer column7;

    private Long column8;
```

3.4.4.3. 字符类型

字段

字段名	#	数据类型	长度	精度	标度	Identity	Collation	非空	默认值	描述
id	1	int4		10				☑	nextval('h...	id
column1	2	bpchar	1	1			default	☐		
column2	3	bpchar					default	☐		bpchar
column3	4	varchar					default	☐		varchar
column4	5	text					default	☐		text
column5	6	name						☐		name
column6	7	bytea						☐		bytea
column7	8	bool	1	1				☐		bool
column8	9	bit		1				☐		bit
column9	10	money						☐		money
column10	11	json						☐		json

约束
外键
索引
Dependencies
参照
触发器
Rules
Statistics
权限
DDL
Virtual

```java
    @Id
    private Integer id;

    private String column1;

    private String column2;

    private String column3;

    private String column4;

    private String column5;

    private byte[] column6;

    private Boolean column7;

    private Integer column8;

    private BigDecimal column9;
```

3.4.5. Hibernate 操作存储过程

（1）首先我们需要在数据库定义一个函数

```
drop function hcj.deleteid
create or replace function hcj.deleteid(in id1 int) returns setof int as
$$
begin
    delete from hcj.test_int where id=id1;
end;
$$language plpgsql;
```

注意点 1：将该函数定义到对应的模式下，否则后续会报错找不到函数

注意点 2：在函数的参数中，应该加上 in,如 in id1 int，表明这是一个徐娅传入的参数

（2）而后在程序中调用

```java
@PersistenceContext
private EntityManager entityManager;
```

```
@Test
public void deleteProcedure()
{
    StoredProcedureQuery query = entityManager.createStoredProcedureQuery( s: "deleteid");
    query.registerStoredProcedureParameter( i: 1, Integer.class, ParameterMode.IN);
    query.setParameter( i: 1, o: 1);
    query.execute();
}
```

步骤 1：定义 EntityManager

步骤 2：由 EntityManager 创建查询

步骤 3：查询的参数传递应先“注册”，即如图所示，第一个参数为 Integer 类型且是一个传入的参数

步骤 4：设置参数，而后执行即可

注：若需要查看参数返回值等，需要在 query.execute()之前，如下：

4. 中间件适配

4.1. Tomcat

Step1:在本地安装 Tomcat 7.0.109

Step2:将 vastbase 驱动包 vastbase-2.0.jar 拷贝进 apache-tomcat-7.0.109\lib

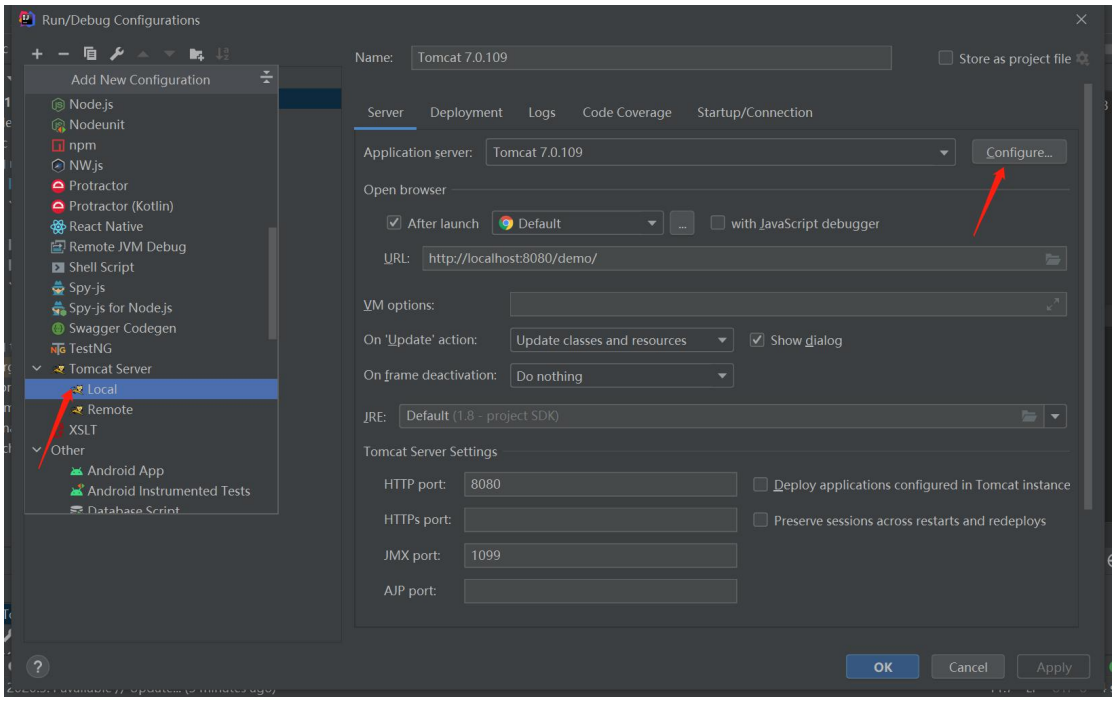
Step3:在 Tomcat 中配置数据源和驱动

打开 apache-tomcat-7.0.109\conf\context.xml 做如下配置:

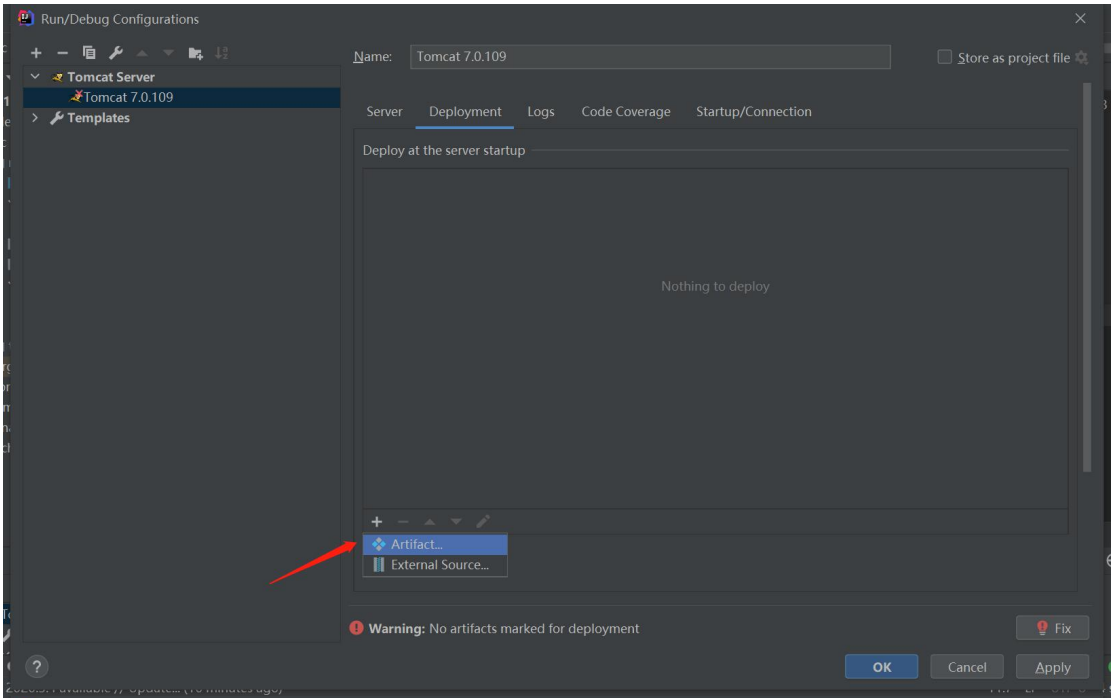
```
<!-- 配置数据源信息及驱动 -->
<Resource name="myDatasource"
 auth="Container"
 type="javax.sql.DataSource"
 maxTotal="100"
 maxIdle="30"
 maxWaitMillis="10000"
 username="[redacted]"
 password="[redacted]"
 driverClassName="org.postgresql.Driver"
 url="jdbc:postgresql://172.16.103.120:5450/test"
 />
```

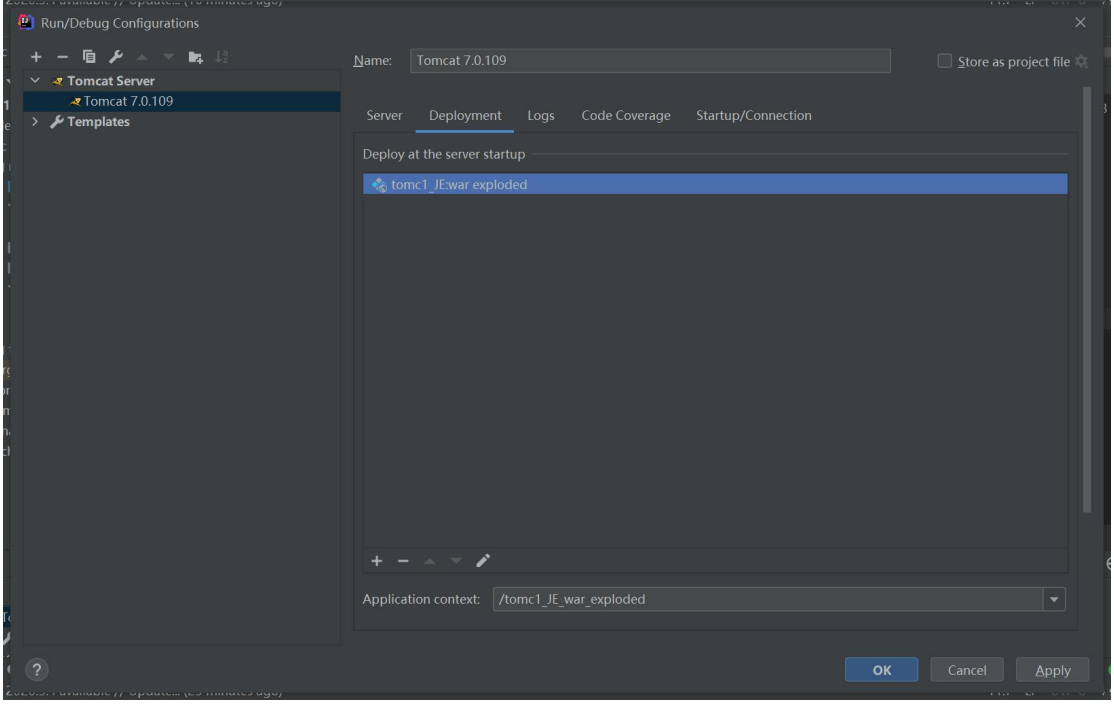
配置 Tomcat 的数据源信息及驱动，其中 name 为自定义的数据源名称

Step4:创建 web 项目，指定本地 Tomcat7.0.109



在 Configuration 中添加 Local Tomcat Server，Configure 选择本地安装的 Tomcat7.0.109





部署项目

Step5:在 web.xml 中配置数据源

```
<resource-ref>
    <description>DB Connection</description>
    <res-ref-name>myDatasource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

其中<res-ref-name>标签对应的是在 context.xml 中配置的数据源名称

Step6:在 JSP 中使用数据库连接池

```
String DSNAME = "java:comp/env/myDatasource";
String SQL = "select * from zy.user";
try {
  Context ctx = new InitialContext();
  DataSource ds = (DataSource) ctx.lookup(DSNAME);
  try {
    Connection conn = ds.getConnection();
    Statement statement = conn.createStatement();
    ResultSet resultSet = statement.executeQuery(SQL);
  } catch (SQLException throwables) {
    throwables.printStackTrace();
  }
} catch (NamingException e) {
  e.printStackTrace();
}
```

其中 DSNAME 中“java:comp/env/myDatasource”是固定写法，表示得到配置环境

4.2. ShardingSphere

4.2.1. ShardingSphere-JDBC

引入 vastbase 的 maven 依赖

```
<dependency>
    <groupId>cn.com.atlasdata</groupId>
    <artifactId>vastbase</artifactId>
    <version>2.0</version>
</dependency>
```

4.2.1.1. 数据分片

Step1:准备好分片后的数据库

Step2:引入 ShardingSphere 的 maven 依赖

```
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
    <version>4.1.1</version>
</dependency>
```

Step3:配置 shardingsphere

1.配置多数据源信息

```yaml
spring:
  shardingsphere:
    dataSource:
      names: ds0,ds1
      ds0:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: org.postgresql.Driver
        url: jdbc:postgresql://172.16.103.120:6028/test
        username: test
        password: Aa123456
      ds1:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: org.postgresql.Driver
        url: jdbc:postgresql://172.16.103.120:6028/test02
        username: test
        password: Aa123456
```

2.配置分片策略

```yaml
    sharding:
      tables:
        t_user:
          # 逻辑表t_user有两个物理表，ds0.t_user0和ds0.t_user1
          actual-data-nodes: ds0.t_user$->{0..1}
          # 配置分表策略，id为偶数的路由到t_user0，id为奇数的路由到t_user1
          table-strategy:
            inline:
              sharding-column: id
              algorithm-expression: t_user$->{id % 2}
        t_customer:
          actual-data-nodes: ds1.t_customer$->{0..1}
          table-strategy:
            inline:
              sharding-column: id
              algorithm-expression: t_customer$->{id % 2}
```

相关数据的 SQL 操作便会被路由到所配置的数据源分片节点上

4.2.1.2. 分布式事务

以 XA 事务为例

Step1:导入 maven 依赖

```xml
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-transaction-xa-core</artifactId>
    <version>4.1.1</version>
</dependency>
```

Step2:编写配置类，向 Spring 容器中注入 PlatformTransactionManager 和 JdbcTemplate

```java
@Configuration
@EnableTransactionManagement
public class TransactionConfiguration {

    @Bean
    public PlatformTransactionManager txManager(final DataSource dataSource){
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean
    public JdbcTemplate jdbcTemplate(final DataSource dataSource) { return new JdbcTemplate(dataSource); }
}
```

Step3:编写业务类，实现数据库操作并使用业务

ShardingSphere 分布式事务注解@ShardingTransactionType 要和 Spring 事务注解 @Transactional 一并使用才会生效；

doInsert 方法向两个数据库中插入数据，之后手动抛出异常，在此之前对两个数据插入的数据都会回滚。

```java
@Transactional
@ShardingTransactionType(TransactionType.XA)
public void insertFailed(final int count){
    jdbcTemplate.execute( sql: "INSERT INTO t_record(id, name) VALUES (?, ?)",
            (PreparedStatementCallback<TransactionType>) preparedStatement -> {
        doInsert(count, preparedStatement);
        throw new SQLException("mock transaction failed");
    });
}
```

若要改成使用 BASE 事务，需要先启动 seata 服务器并导入 Sharding-BASE 依赖

```xml
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-transaction-base-seata-at</artifactId>
    <version>4.1.1</version>
</dependency>
```

4.2.1.3. 读写分离

Step1:导入 maven 依赖

```xml
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
    <version>4.1.1</version>
</dependency>
```

Step2:配置数据源

配置多数据源，指定主库与从库。4.1.1 版本支持单主库多从库，不支持主从库之间的数据同步。

当一个线程中没有进行主库的修改，读操作会被路由到从库进行查询；若一个线程中主库进行了修改，为了保持数据一致性，读操作也会被路由到主库查询。

```yaml
spring:
  shardingsphere:
    # 配置数据源信息
    datasource:
      names: master,slave0
      master:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: org.postgresql.Driver
        url: jdbc:postgresql://172.16.03.120/db0
        username: test_zy
        password: Aa123456
      slave0:
        type: com.alibaba.druid.pool.DruidDataSource
        driver-class-name: org.postgresql.Driver
        url: jdbc:postgresql://172.16.03.120/db1
        username: test_zy
        password: Aa123456
    masterslave:
      name: ms
      # 指定主库与从库
      master-data-source-name: master
      slave-data-source-names: slave0
```

4.2.2. ShardingSphere-Proxy

首先需要下载并解压 apache-shardingsphere-4.1.1-sharding-proxy-bin.tar

由于 ShardingSphere-Proxy 默认只支持 postgresql，使用其他数据库时要先将对应 java 驱动包加入 lib 目录，因此将 vastbase-2.0 驱动包加入 lib 目录。

4.2.2.1. 数据分片

在/conf/config-sharding.yaml 配置数据源及数据分片策略

Step1:配置数据源

```yaml
dataSources:
  ds_0:
    url: jdbc:postgresql://172.16.03.120:7420/db0?serverTimezone=UTC&useSSL=false
    username: root
    password: 123456
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 50
  ds_1:
    url: jdbc:postgresql://172.16.03.120:7420/db1?serverTimezone=UTC&useSSL=false
    username: root
    password: 123456
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 50
```

Step2:配置分片策略

```yaml
shardingRule:
  tables:
    t_order:
      actualDataNodes: ds_${0..1}.t_order_${0..1}
      tableStrategy:
        inline:
          shardingColumn: order_id
          algorithmExpression: t_order_${order_id % 2}
      keyGenerator:
        type: SNOWFLAKE
        column: order_id
  defaultDatabaseStrategy:
    inline:
      shardingColumn: user_id
      algorithmExpression: ds_${user_id % 2}
  defaultDataSourceName: ds_0
```

Step3:配置 server.yaml

在/conf/server.yaml 配置代理数据库的用户、密码及逻辑数据库权限

Step4:启动 shardingsphere-proxy 服务

windows 运行：/bin/start.bat

linux 运行：/bin/start.sh

Step5:使用代理数据库

使用任何 PostgreSQL 客户端，与代理数据库建立连接，即可当作普通数据库来使用

用户及密码：server.yaml 中的配置

Host：127.0.0.1

Port：3307

4.2.2.2. 读写分离

在/conf/config-master_slave.yaml 配置数据源及读写分离主从库

Step1:配置数据源

```yaml
dataSources:
  master_ds:
    url: jdbc:postgresql://172.16.03.120:7280/db0?serverTimezone=UTC&useSSL=false
    username: root
    password: 123456
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 50
  slave_ds_0:
    url: jdbc:postgresql://172.16.03.120:7280/db1?serverTimezone=UTC&useSSL=false
    username: root
    password: 123456
    connectionTimeoutMilliseconds: 30000
    idleTimeoutMilliseconds: 60000
    maxLifetimeMilliseconds: 1800000
    maxPoolSize: 50
```

Step2:配置主从库

指定主库与从库，并将主-从数据库指定为默认数据库

```yaml
shardingRule:
  masterSlaveRules:
    ds_0:
      masterDataSourceName: master_ds
      slaveDataSourceNames:
        - slave_ds_0
  defaultDataSourceName: ds_0
```

Step3:配置 server.yaml

在/conf/server.yaml 配置代理数据库的用户、密码及逻辑数据库权限

Step4:启动 shardingsphere-proxy 服务

windows 运行：/bin/start.bat

linux 运行：/bin/start.sh

Step5:使用代理数据库

使用任何 PostgreSQL 客户端，与代理数据库建立连接，即可当作普通数据库来使用

用户及密码：server.yaml 中的配置

Host：127.0.0.1

Port：3307

电话：010-82838118
地址：北京市海淀区学院路 30 号科大天工大厦 B 座 6 层
官网：www.vastdata.com.cn