#### 封面01.jpg

**MySQL主从复制讲义**

# MySQL主从复制概述

## 主从复制概述

在实际生产中，数据的重要性不言而喻

如果我们的数据库只有一台服务器，那么很容易产生单点故障的问题，比如这台服务器访问压力过大而没有响应或者奔溃，那么服务就不可用了，再比如这台服务器的硬盘坏了，那么整个数据库的数据就全部丢失了，这是重大的安全事故.

为了避免服务的不可用以及保障数据的安全可靠性，我们至少需要部署两台或两台以上服务器来存储数据库数据，也就是我们需要将数据复制多份部署在多台不同的服务器上，即使有一台服务器出现故障了，其他服务器依然可以继续提供服务.

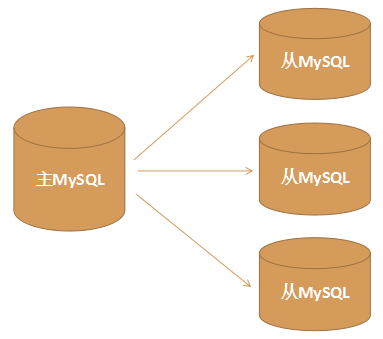
MySQL提供了主从复制功能以提高服务的可用性与数据的安全可靠性.

主从复制是指服务器分为主服务器和从服务器，主服务器负责读和写，从服务器只负责读，主从复制也叫 master/slave，master是主，slave是从，但是并没有强制，也就是说从也可以写，主也可以读，只不过一般我们不这么做。

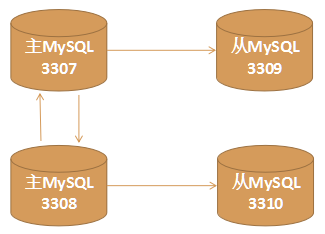
主从复制可以实现对数据库备份和读写分离

## 主从复制架构

### 一主多从架构



### 双主双从架构

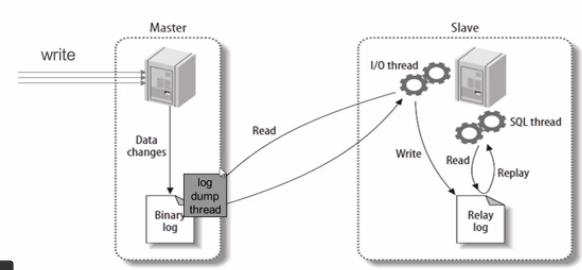


## 主从复制原理（流程|步骤）

* 当 master 主服务器上的数据发生改变时，则将其改变写入二进制事件日志文件中
* salve 从服务器会在一定时间间隔内对 master 主服务器上的二进制日志进行探测，探测其是否发生过改变，如果探测到 master 主服务器的二进制事件日志发生了改变，则开始一个 I/O Thread 请求 master 二进制事件日志
* 同时 master 主服务器为每个 I/O Thread 启动一个dump Thread，用于向其发送二进制事件日志
* slave 从服务器将接收到的二进制事件日志保存至自己本地的中继日志文件中
* salve 从服务器将启动 SQL Thread 从中继日志中读取二进制日志，在本地重放，使得其数据和主服务器保持一致；
* 最后 I/O Thread 和 SQL Thread 将进入睡眠状态，等待下一次被唤醒

**注意：主从复制的过程会有很小的延迟，基本没有影响**

## MySQL主从复制流程图



# MySQL多实例搭建

## 多实例概述

MySQL多实例是指安装MySQL之后，在一台Linux服务器上同时启动多个MySQL数据库（实例），不需要安装多个MySQL（适合技术研究和学习的场景）

如果是有多台Linux服务器，那么我们需要每台服务器都分别安装MySQL（适用于实际线上生产环境）

我们此处计划在一台Linux上启动多个MySQL，这样适合我们的技术研究和学习，如果要在多台Linux分别启动MySQL，这个与我们在一台机器上的配置与操作都是完全一样的。

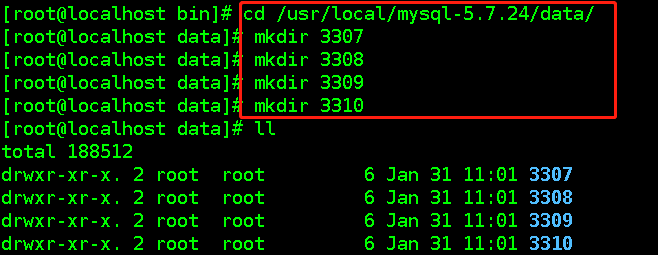
**如何实现在一台Linux服务器上同时启动多个MySQL数据库（实例）？**

通过为各个数据库实例配置独立的配置文件来实现，即每个数据库实例有自己单独的配置文件

## 多实例配置

### 在MySQL安装主目录下创建四个实例存放数据的目录

/data/3307, /data/3308, /data/3309, /data/3310



### 执行数据库初始化

在MySQL的**/usr/local/mysql-5.7.24/bin**目录下执行命令

./mysqld --initialize-insecure --basedir=/usr/local/mysql-5.7.24 --datadir=/usr/local/mysql-5.7.24/data/3307 --user=mysql

./mysqld --initialize-insecure --basedir=/usr/local/mysql-5.7.24 --datadir=/usr/local/mysql-5.7.24/data/3308 --user=mysql

./mysqld --initialize-insecure --basedir=/usr/local/mysql-5.7.24 --datadir=/usr/local/mysql-5.7.24/data/3309 --user=mysql

./mysqld --initialize-insecure --basedir=/usr/local/mysql-5.7.24 --datadir=/usr/local/mysql-5.7.24/data/3310 --user=mysql

**initialize-insecure 表示不生成MySQL数据库root用户的随机密码，即root密码为空**

### 配置四个MySQL数据库服务的my.cnf文件

#### 为了方便操作，在Xshell中打开4个选项卡，去连接不同的是MySQL实例



#### 在/data/3307, /data/3308, /data/3309, /data/3310 四个目录下分别创建一个my.cnf文件，在四个my.cnf文件分别配置如下内容

**注意：不同的实例下配置要修改端口号**

[client]  
port = 3307  
socket = /usr/local/mysql-5.7.24/data/3307/mysql.sock  
default-character-set=utf8  
  
[mysqld]  
port = 3307  
socket = /usr/local/mysql-5.7.24/data/3307/mysql.sock  
datadir = /usr/local/mysql-5.7.24/data/3307  
log-error = /usr/local/mysql-5.7.24/data/3307/error.log  
pid-file = /usr/local/mysql-5.7.24/data/3307/mysql.pid  
  
character-set-server=utf8  
lower\_case\_table\_names=1  
autocommit = 1

## 多实例启动

切换到/usr/local/mysql-5.7.24/bin目录下，使用 msyqld\_safe 命令指定配置文件并启动MySQL服务：

./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3307/my.cnf &  
./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3308/my.cnf &  
./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3309/my.cnf &  
./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3310/my.cnf &

**其中 --defaults-file 是指定配置文件，& 符合表示后台启动**

## 启动后配置

### 登录进入每一个MySQL实例

在mysql-5.7.24/bin目录下执行命令

#### 方式一：使用用端口、主机登录（推荐使用该方式）

./mysql -uroot -p -P3307 -h127.0.0.1

#### 方式二：使用socket文件

./mysql -uroot -p -S /usr/local/mysql-5.7.24/data/3307/mysql.sock

-p 是指定密码，如果没有密码则可以不写 -p，

-S是指定sock文件，mysql.sock文件是服务器与本机客户端进行通信的ip与端口文件

### 修改mysql的密码

alter user 'root'@'localhost' identified by '123456';

其中123456是我们设置的密码

### 授权远程访问（这样远程客户端才能访问）

grant all privileges on \*.\* to root@'%' identified by '123456';

其中\*.\* 的第一个\*表示所有数据库名，第二个\*表示所有的数据库表

root@'%' 中的root表示用户名

%表示所有ip地址，%也可以指定具体的ip地址，比如root@localhost，root@192.168.10.129

### 刷新权限

flush privileges;

## 多实例关闭

切换到/usr/local/mysql-5.7.24/bin目录下，使用 mysqladmin 命令 shutdown

#### 方式一：使用端口、主机关闭 （推荐）

./mysqladmin -uroot -p -P3307 -h127.0.0.1 shutdown

#### 方式二:使用socket文件

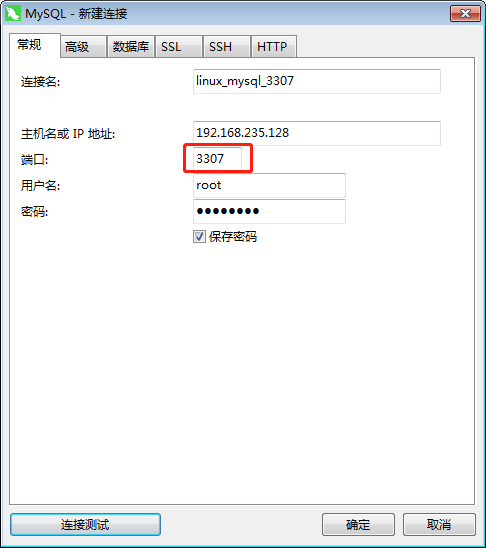
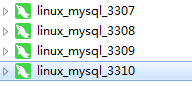
./mysqladmin -uroot -p -S /usr/local/mysql-5.7.24/data/3307/mysql.sock shutdown

#### 方式三: 进入MySQL的客户端命令行，直接执行 shutdown

#### 退出MySQL命令行

在客户端命令行下执行 exit

## 在Navicat中创建多个连接，分别连接主从库

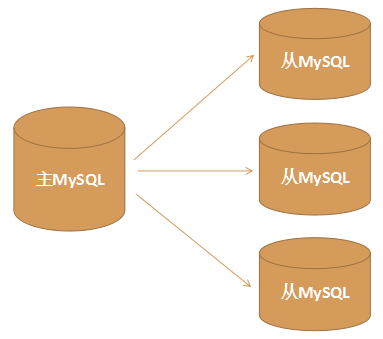
 

# 一主多从环境搭建

## 概述

当系统对数据的读取比较多时，为了分摊读的压力，可以采用一主多从架构，实现读写分离

## 一主三从架构架构图



## 环境配置（先将各个服务器停掉,删除掉中文注释）

### 在MySQL主服务器(3307)配置文件my.cnf里面加入

log-bin=mysql-bin    #表示启用二进制日志

server-id=3307  #表示server编号，编号要唯一

### 在MySQL从服务器(3308)配置文件my.cnf里面加入

server-id=3308 #表示server编号，编号要唯一

### 在MySQL从服务器(3309)配置文件my.cnf里面加入

server-id=3309 #表示server编号，编号要唯一

### 在MySQL从服务器(3310)配置文件my.cnf里面加入

server-id=3310 #表示server编号，编号要唯一

## 服务启动

**进入/usr/local/mysql-5.7.24/bin目录，重启四个MySQL服务，启动时指定配置文件**

./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3307/my.cnf &

./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3308/my.cnf &

./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3309/my.cnf &

./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3310/my.cnf &

## 主从设置（重要）

### 主服务器（3307）设置

**需要登录到主服务器3307的客户端**

**在/usr/local/mysql-5.7.24/bin目录下执行 ./mysql -uroot -p -P3307 -h127.0.0.1**

#### 在主服务器上创建复制数据的账号并授权

grant replication slave on \*.\* to 'copy'@'%' identified by '123456';

注意：该语句可完成授权、创建用户、修改密码操作

#### 查看主服务器状态

show master status;

mysql主服务器默认初始值：

File：mysql-bin.000001

Position：154

#### 如果主服务状态不是初始状态，需要重置状态

执行命令：reset master;



### 从服务器（3308|3309|3310）设置

**需要登录到主服务器3308|3309|3310的客户端**

**在/usr/local/mysql-5.7.24/bin目录下执行**

**./mysql -uroot -p -P3308|3309|3310 -h127.0.0.1**

#### 查看从服务器状态

show slave status;

初始状态：Empty set

#### 如果从服务器不是初始状态，建议重置一下

stop slave; #停止复制，相当于终止从服务器上的IO和SQL线程

reset slave;

#### 设置从服务器的master

在从服务器客户端执行

change master to master\_host='192.168.235.128',master\_user='copy',  
master\_port=3307,master\_password='123456',  
master\_log\_file='mysql-bin.000001',master\_log\_pos=154;

#### 在从机器上执行开始复制命令

start slave;

#### 至此一个一主多从搭建完成

## 主从验证

### 检查从服务器复制功能状态

在从服务器的客户端执行以下命令：show slave status \G;

\G表示格式化输出

如果 Slave\_IO\_Running 和 Slave\_SQL\_Running 均为 YES，则表示主从关系正常

### 在主服务器上创建数据库、表、数据，然后在从服务器上查看是否已经复制

* 在Navicat主服务器上创建库test，查看从服务器情况
* 在Navicat主服务器上test库中创建表user(id,name)，查看从服务器情况
* 在Navicat主服务器上test库的user中添加数据，查看从服务器情况
* 在Navicat主服务器上test库的user中修改数据，查看从服务器情况

**如果以上操作过程若显示正常，则主从服务器配置完成**

### 查看主从复制binlog日志文件内容

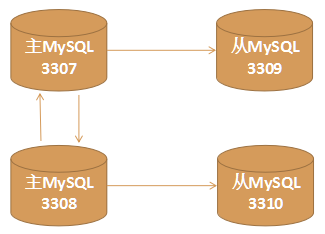
在主服务器客户端执行：show binlog events in 'mysql-bin.000001'\G;

# 双主双从环境搭建

## 概述

一主多从，可以缓解读的压力，但是一旦主宕机了，就不能写了,所以我们可以采用双主双从架构来改进它的不足。

## 双主双重架构图



**架构规划**

* 主master 3307 ---> 从slave 3309
* 主master 3308 ---> 从slave 3310
* 3307 <---> 3308 互为主从
* 2个写节点，每个写节点下又是2个读节点

## 环境配置（不要含有中文注释）

### 在MySQL主服务器(3307)配置文件my.cnf里面加入（已做）

log-bin=mysql-bin

server-id=3307

### 在MySQL主服务器(3308)配置文件my.cnf里面加入

log-bin=mysql-bin

server-id=3308

### 在MySQL从服务器(3309)配置文件my.cnf里面加入（已做）

server-id=3309

### 在MySQL从服务器(3310)配置文件my.cnf里面加入（已做）

server-id=3310

### 在第一台主服务器3307的my.cnf文件增加如下配置

auto\_increment\_increment=2

auto\_increment\_offset=1 #不一样的点 相当于起始值

log-slave-updates

sync\_binlog=1

### 在第二台主服务器3308的my.cnf文件增加如下配置

auto\_increment\_increment=2

auto\_increment\_offset=2 #不一样的点 相当于起始值

log-slave-updates

sync\_binlog=1

### 配置项说明

#### auto\_increment\_increment

控制主键自增的自增步长，用于防止Master与Master之间复制出现重复自增字段值，通常auto\_increment\_increment=n，有多少台主服务器，n 就设置为多少

#### auto\_increment\_offset＝1

设置自增起始值，这里设置为1，这样Master的auto\_increment字段产生的数值是：1, 3, 5, 7, …等奇数ID

注意auto\_increment\_offset的设置，不同的master设置不应该一样，否则就容易引起主键冲突，比如master1的offset=1，则master2的offset=2，master3的offset=3

#### log-slave-updates

在双主模式中，log-slave-updates 配置项一定要配置，否则在master1（3307）上进行了更新数据，在master2（3308）和slave1（3309）上会更新，但是在slave2（3310）上不会更新

#### sync\_binlog

表示每几次事务提交，MySQL把binlog缓存刷进日志文件中，默认是0，最安全的是设置为1

sync\_binlog=0，当事务提交之后，MySQL不做fsync之类的磁盘同步指令刷新binlog\_cache中的信息到磁盘，而让Filesystem自行决定什么时候来做同步，或者cache满了之后才同步到磁盘。

sync\_binlog=n，当每进行n次事务提交之后，MySQL将进行一次fsync之类的磁盘同步指令来将binlog\_cache中的数据强制写入磁盘。

#### 注意

* 从库只开启log-bin功能，不添加log-slave-updates参数，从库从主库复制的数据不会写入log-bin日志文件里。
* 开启log-slave-updates参数后，从库从主库复制的数据会写入log-bin日志文件里。这也是该参数的功能。
* 直接向从库写入数据时，是会写入log-bin日志的
* 在自动生成主键的时候，会在已生成主键的基础上按照规则生成，即比存在的值大

## 服务启动

**进入/usr/local/mysql-5.7.24/bin目录，重启四个MySQL服务，启动时指定配置文件**

./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3307/my.cnf &

./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3308/my.cnf &

./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3309/my.cnf &

./mysqld\_safe --defaults-file=/usr/local/mysql-5.7.24/data/3310/my.cnf &

## 主从设置（重要）

### 主服务器（3307|3308）设置

**需要登录到主服务器3307|3308的客户端**

**在/usr/local/mysql-5.7.24/bin目录下执行 ./mysql -uroot -p -P3307|3308 -h127.0.0.1**

#### 在两台主服务器（3307|3308）上创建复制数据的账号并授权

3307已经做过，只需要在3308上执行即可

grant replication slave on \*.\* to 'copy'@'%' identified by '123456';

#### 在两台主服务器（3307|3308）上停止复制并重置服务器状态

show master status;

mysql主服务器默认初始值：

File：mysql-bin.000001

Position：154

#### 如果主服务状态不是初始状态，需要重置状态

执行命令：reset master;



#### 因为之前3308机器设置过从，所以该3308机器应该执行

stop slave;

reset slave;

### 从服务器（3307|3308|3309|3310）设置

**需要登录到主服务器3308|3309|3310的客户端**

**在/usr/local/mysql-5.7.24/bin目录下执行**

**./mysql -uroot -p -P3308|3309|3310 -h127.0.0.1**

#### 在从服务器上（3309、3310）停止复制并重置服务器状态：

stop slave;

reset slave;

#### 设置从服务器的master（相当于是4台都需要设置）

**注意：这里的IP和端口要根据自己的实际情况修改**

##### 设置从服务器3308、3309的主，他们的主均为3307

即从服务器3308和3309客户端上执行如下操作

change master to master\_host='192.168.235.128',master\_user='copy',  
master\_port=3307,master\_password='123456',  
master\_log\_file='mysql-bin.000001',master\_log\_pos=154;

##### 设置从服务器3307、3310的主，他们的主均为3308

即从服务器3307和3310客户端上执行如下操作

change master to master\_host='192.168.235.128',master\_user='copy',  
master\_port=3308,master\_password='123456',  
master\_log\_file='mysql-bin.000001',master\_log\_pos=154;

#### 在从机器上执行开始复制命令(4台MySQL上都执行)

start slave;

#### 至此双主双从就搭建好了

## 双主双从验证

### 检查从服务器复制功能状态

在从服务器的客户端执行以下命令：show slave status \G;

\G表示格式化输出

如果 Slave\_IO\_Running 和 Slave\_SQL\_Running 均为 YES，则表示主从关系正常

### 在主服务器上操作测试数据的复制和同步情况

# 多数据源问题

多数据源问题是指在一个项目工程中，需要连接多个数据库

以上我们配置了数据库的主从,在实际开发中可能会有以下几种情况

* 读写分离:项目中使用多数据源
* 项目中只操作主库(读和写),主从只是起到了备份作用,程序员不需要关心主从结构

## 常见的多数据源技术选型模式

* JDBC

项目是JDBC开发，比较老

编写多个getConnection方法，分别连接不同的数据库即可

* Hibernate

项目是SSH框架开发，现在项目使用少一些

Spring + Hibernate

Springboot + Hibernate

* Mybatis

项目是SSM框架开发

**Spring + Mybatis**

**Springboot + Mybatis**

Spring + JPA

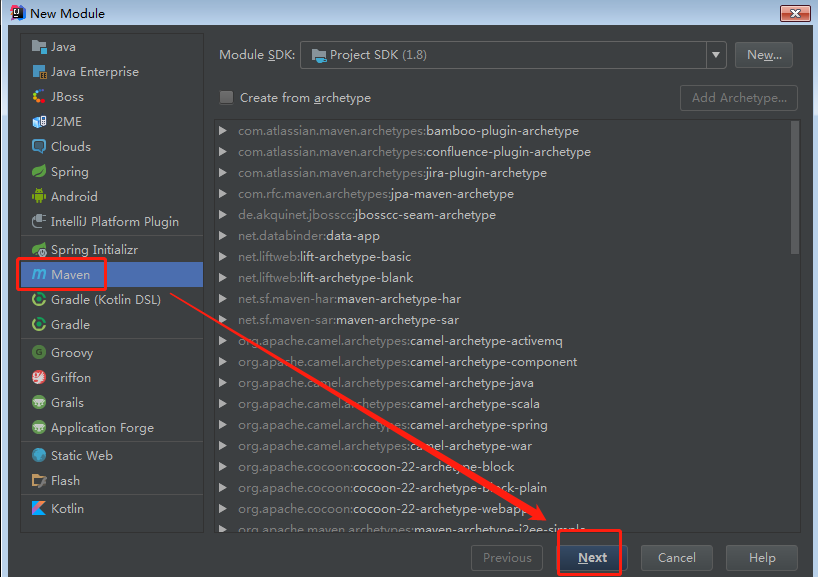
Springboot + JPA

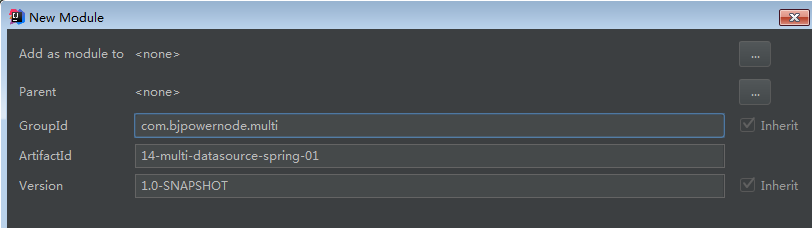
我们这里主要给大家介绍Spring+Mybatis和SpringBoot+Mybatis开发模式下的多数据源问题

## Spring+Mybatis方案一实现步骤

**核心思想：基于Mapper包的隔离，每个Mapper包操作不同的数据库，每个Mapper包对应一个数据库**

### 创建一个普通的maven项目14-multi-datasource-spring-01





### 在pom.xml文件中添加相关的依赖

<!--Spring相关的依赖-->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-jdbc</artifactId>  
 <version>5.1.4.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-tx</artifactId>  
 <version>5.1.4.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.1.4.RELEASE</version>  
</dependency>  
  
<!--Mybatis相关依赖-->  
<!--Mybatis框架依赖-->  
<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.4.6</version>  
</dependency>  
<!--Mybatis与Spring整合依赖-->  
<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis-spring</artifactId>  
 <version>1.3.2</version>  
</dependency>  
<!--MySQL数据库连接驱动 版本不要过高-->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.43</version>  
</dependency>  
<!--JDBC 数据库连接池-->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.1.1</version>  
</dependency>

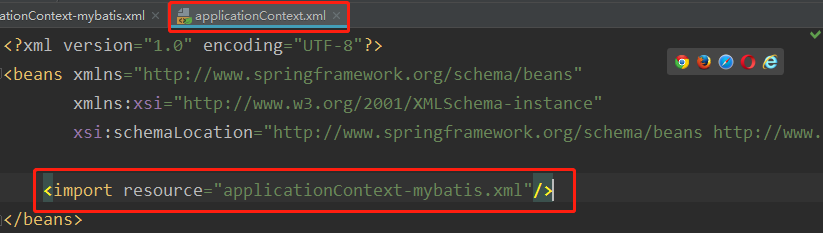
### 在pom.xml文件添加resource，指定编译Mybatis映射文件

<resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
</resource>

### 创建Mybatis整合多数据源的配置文件，applicationContext-mybatis.xml



### 创建Spring的核心配置文件applicationContext.xml，引入Mybatis配置文件



### 在applicationContext-mybatis.xml配置3307|3308|3309|3310这四个数据源

<bean id="dataSource3307" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3307/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
</bean>  
<bean id="dataSource3308" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3308/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
</bean>  
<bean id="dataSource3309" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3309/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
</bean>  
<bean id="dataSource3310" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3310/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
</bean>

### 在applicationContext-mybatis.xml配置四个sessionFactory，分别使用3307|3308|3309|3310这四个数据源

<!--3307数据库的sessionFactory-->  
<bean id="sessionFactory3307" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource3307"/>  
</bean>  
<!--3308数据库的sessionFactory-->  
<bean id="sessionFactory3308" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource3308"/>  
</bean>  
<!--3309数据库的sessionFactory-->  
<bean id="sessionFactory3309" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource3309"/>  
</bean>  
<!--3310数据库的sessionFactory-->  
<bean id="sessionFactory3310" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource3310"/>  
</bean>

### 在applicationContext-mybatis.xml配置四个Mapper包的扫描，分别扫描3307|3308|3309|3310

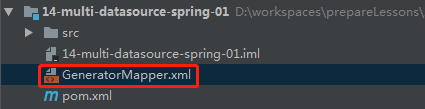
也就是说每个mapper下的接口只能操作一个数据库

<!--扫描3307库对应的mapper包，也就是说该Mapper下的接口操作的是3307数据库-->  
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <property name="sqlSessionFactoryBeanName" value="sessionFactory3307"/>  
 <property name="basePackage" value="com.bjpowernode.multi.mapper.mapper3307"/>  
</bean>  
<!--扫描3308库对应的mapper包，也就是说该Mapper下的接口操作的是3308数据库-->  
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <property name="sqlSessionFactoryBeanName" value="sessionFactory3308"/>  
 <property name="basePackage" value="com.bjpowernode.multi.mapper.mapper3308"/>  
</bean>  
<!--扫描3309库对应的mapper包，也就是说该Mapper下的接口操作的是3309数据库-->  
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <property name="sqlSessionFactoryBeanName" value="sessionFactory3309"/>  
 <property name="basePackage" value="com.bjpowernode.multi.mapper.mapper3309"/>  
</bean>  
<!--扫描3310库对应的mapper包，也就是说该Mapper下的接口操作的是3310数据库-->  
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <property name="sqlSessionFactoryBeanName" value="sessionFactory3310"/>  
 <property name="basePackage" value="com.bjpowernode.multi.mapper.mapper3310"/>  
</bean>

### 在pom.xml文件中加入Mybatis反向工程插件

<!--mybatis代码自动生成插件-->  
<plugin>  
 <groupId>org.mybatis.generator</groupId>  
 <artifactId>mybatis-generator-maven-plugin</artifactId>  
 <version>1.3.7</version>  
 <configuration>  
 <!--配置文件的位置-->  
 <configurationFile>GeneratorMapper.xml</configurationFile>  
 <verbose>true</verbose>  
 <overwrite>true</overwrite>  
 </configuration>  
</plugin>

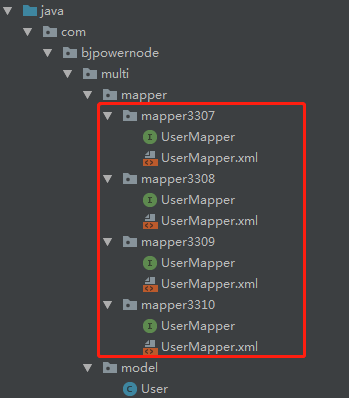
### 将反向工程的配置文件拷贝到当前项目中



### 修改反向工程配置文件，将3307数据库中的User表反向生成到com.bjpowenrode.multi包下



### 在mapper包下，分别创建子包mapper3307| mapper3308| mapper3309| mapper3310，并将生成的代码拷贝到不同的这几个子包下，然后删除掉mapper包下的生成的接口及xml文件



### 修改子包下的xml文件的命名空间为对应的包



### 在com.bjpowernode.multi.service包下创建UserService接口，在接口中提供getUserByIdFrom3307|3308|3309|3310方法

public interface UserService {  
 User getUserByIdFrom3307(Integer id);  
 User getUserByIdFrom3308(Integer id);  
 User getUserByIdFrom3309(Integer id);  
 User getUserByIdFrom3310(Integer id);  
}

### 在com.bjpowernode.multi.service.impl包下创建UserServiceImpl实现类，并实现接口中的方法

因为我们现在4个数据库都是相同的表（实际不会存在这种情况），所以在service实现类中注入Mapper接口的时候，需要使用全限定名进行区分，并且变量名也不能一样

@Service  
public class UserServiceImpl implements UserService{  
 @Autowired  
 private com.bjpowernode.multi.mapper.mapper3307.UserMapper userMapper3307;  
  
 @Autowired  
 private com.bjpowernode.multi.mapper.mapper3308.UserMapper userMapper3308;  
  
 @Autowired  
 private com.bjpowernode.multi.mapper.mapper3309.UserMapper userMapper3309;  
  
 @Autowired  
 private com.bjpowernode.multi.mapper.mapper3310.UserMapper userMapper3310;  
  
 public User getUserByIdFrom3307(Integer id) {  
 return userMapper3307.selectByPrimaryKey(id);  
 }  
  
 public User getUserByIdFrom3308(Integer id) {  
 return userMapper3308.selectByPrimaryKey(id);  
 }  
  
 public User getUserByIdFrom3309(Integer id) {  
 return userMapper3309.selectByPrimaryKey(id);  
 }  
  
 public User getUserByIdFrom3310(Integer id) {  
 return userMapper3310.selectByPrimaryKey(id);  
 }  
}

### 在mapper3307|3308|3309|3310包下接口上的加@Component(“userMapper3307|3308|3309|3310”)

因为默认情况下，Spring容器创建的Mapper接口的代理对象名是接口名首字母小写，因为我们现在操作相同的表，所以接口名一样，默认生成的代理对象的名字也一样，会冲突，需要我们改一下，名字可以随意取，只要不冲突就行，因为@Autowired是根据类型注入的



### 在applicationContext.xml文件中扫描impl包

<context:component-scan base-package="com.bjpowernode.multi.service.impl"/>

### 创建测试类，在main方法获取获取service，调用方法

为了区分效果，可以讲从库的名字修改一下

public class Test {  
 public static void main(String[] args) {  
 ClassPathXmlApplicationContext context =  
 new ClassPathXmlApplicationContext("classpath:applicationContext.xml");  
 UserService userService = context.getBean("userServiceImpl",UserService.class);  
 System.*out*.println("3307数据库：" +userService.getUserByIdFrom3307(4).getName());  
 System.*out*.println("3308数据库：" +userService.getUserByIdFrom3308(3).getName());  
 System.*out*.println("3309数据库：" +userService.getUserByIdFrom3309(4).getName());  
 System.*out*.println("3310数据库：" +userService.getUserByIdFrom3310(4).getName());  
 }  
}

## Spring+Mybatis方案一加事务管理步骤

### 在applicationContext-mybatis.xml文件中为每一个数据源加事务管理器，并开启事务注解

<!--事务管理器-->  
<bean id="transactionManager3307" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource3307"/>  
</bean>  
<tx:annotation-driven transaction-manager="transactionManager3307"/>  
  
<bean id="transactionManager3308" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource3308"/>  
</bean>  
<tx:annotation-driven transaction-manager="transactionManager3308"/>  
  
<bean id="transactionManager3309" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource3309"/>  
</bean>  
<tx:annotation-driven transaction-manager="transactionManager3309"/>  
  
<bean id="transactionManager3310" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource3310"/>  
</bean>  
<tx:annotation-driven transaction-manager="transactionManager3310"/>

### 在UserService接口上加更新用户方法

int updateUserFrom3307(User user);  
int updateUserFrom3308(User user);  
int updateUserFrom3309(User user);  
int updateUserFrom3310(User user);

### 在UserServiceImpl实现类中实现更新用户方法，并在该方法上加事务注解

@Transactional(transactionManager = "transactionManager3307")  
public int updateUserFrom3307(User user) {  
 int updateRow = userMapper3307.updateByPrimaryKeySelective(user);  
 System.*out*.println(updateRow);  
 int a = 10/0;  
 return updateRow;  
}  
@Transactional(transactionManager = "transactionManager3308")  
public int updateUserFrom3308(User user) {  
 int updateRow = userMapper3308.updateByPrimaryKeySelective(user);  
 System.*out*.println(updateRow);  
 int a = 10/0;  
 return updateRow;  
}  
@Transactional(transactionManager = "transactionManager3309")  
public int updateUserFrom3309(User user) {  
 int updateRow = userMapper3309.updateByPrimaryKeySelective(user);  
 System.*out*.println(updateRow);  
 int a = 10/0;  
 return updateRow;  
}  
@Transactional(transactionManager = "transactionManager3310")  
public int updateUserFrom3310(User user) {  
 int updateRow = userMapper3310.updateByPrimaryKeySelective(user);  
 System.*out*.println(updateRow);  
 int a = 10/0;  
 return updateRow;  
}

### 在Test类中进行测试

为了避免同步，我们这里拿从库进行测试

User user = new User();  
user.setId(3);  
user.setName("update-3309");  
userService.updateUserFrom3309(user);

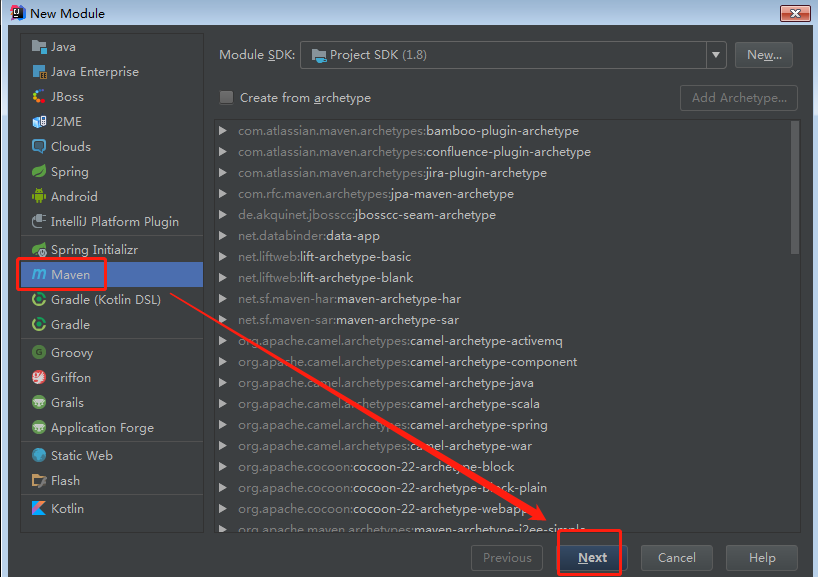
* 发生除0异常，事务回滚，更新失败
* 如果将实现类的方法注解注释，虽然发生异常，依然可以更新成功

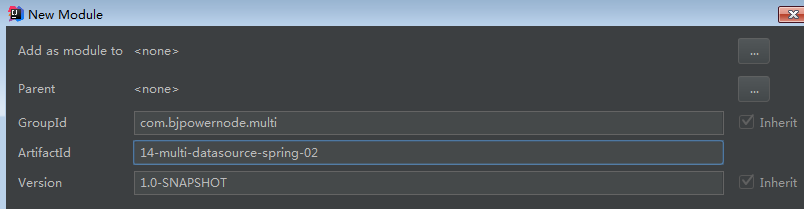
//@Transactional(transactionManager = "transactionManager3309")

## Spring+Mybatis方案二实现步骤

**核心思想：基于动态数据源，在运行的时候才知道要是用哪个数据源**

### 创建一个普通的maven项目14-multi-datasource-spring-02





### 在pom.xml文件中添加相关的依赖

<!--Spring相关的依赖-->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-jdbc</artifactId>  
 <version>5.1.4.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-tx</artifactId>  
 <version>5.1.4.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.1.4.RELEASE</version>  
</dependency>  
  
<!--Mybatis相关依赖-->  
<!--Mybatis框架依赖-->  
<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.4.6</version>  
</dependency>  
<!--Mybatis与Spring整合依赖-->  
<dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis-spring</artifactId>  
 <version>1.3.2</version>  
</dependency>  
<!--MySQL数据库连接驱动 版本不要过高-->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.43</version>  
</dependency>  
<!--JDBC 数据库连接池-->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.1.1</version>  
</dependency>

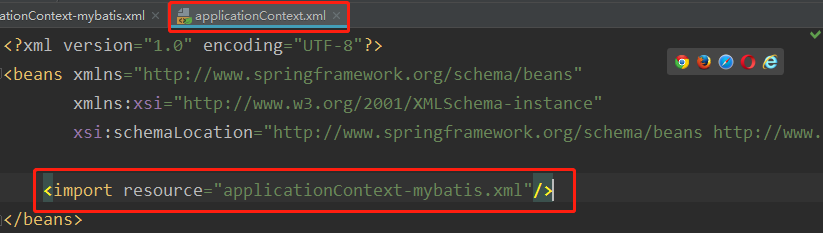
### 在pom.xml文件添加resource，指定编译Mybatis映射文件

<resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
</resource>

### 创建Mybatis整合多数据源的配置文件，applicationContext-mybatis.xml



### 创建Spring的核心配置文件applicationContext.xml，引入Mybatis配置文件



### 在applicationContext-mybatis.xml配置3307|3308|3309|3310这四个数据源

<bean id="dataSource3307" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3307/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
</bean>  
<bean id="dataSource3308" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3308/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
</bean>  
<bean id="dataSource3309" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3309/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
</bean>  
<bean id="dataSource3310" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3310/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
</bean>

### 在applicationContext-mybatis.xml配置一个sessionFactory，引用动态的数据源

动态数据源就是在运行的时候决定采用哪个数据源，在配置文件中不能写死

<!--数据库的sessionFactory-->  
<bean id="sessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dynamicDataSource"/>  
</bean>

**注意：我们现在只引用了dynamicDataSource，但是这个类里具体内容是什么还不知道**

### 在com.bjpowernode.multi.dynamic包下创建DynamicDataSource类继承AbstractRoutingDataSource

#### 设计的思路是将所有数据源放到一个map集合中，通过指定map集合的key，可以动态获取不同的数据源

#### 该类中有个抽象方法determineCurrentLookupKey需要实现

protected Object determineCurrentLookupKey() {  
 return null;  
}

#### 定义几个常量，作为动态数据源对应的key

public static final String *DATASOURCE\_KEY\_3307* = "3307";  
public static final String *DATASOURCE\_KEY\_3308* = "3308";  
public static final String *DATASOURCE\_KEY\_3309* = "3309";  
public static final String *DATASOURCE\_KEY\_3310* = "3310";

#### 如何将指定的key和不同的数据源放到map集合中呢？

在applicationContext-mybatis.xml对动态数据源DynamicDataSource类进行配置

<!--动态数据源-->  
<bean id="dynamicDataSource" class="com.bjpowernode.multi.dynamic.DynamicDataSource">  
 <property name="targetDataSources">  
 <map>  
 <entry key="3307" value-ref="dataSource3307"/>  
 <entry key="3308" value-ref="dataSource3308"/>  
 <entry key="3309" value-ref="dataSource3309"/>  
 <entry key="3310" value-ref="dataSource3310"/>  
 </map>  
 </property>  
</bean>

#### 如何为determineCurrentLookupKey()动态的指定key呢

在进行数据库操作前，我们提前设置使用哪个数据库，然后在指定的数据源的时候，会自动调用这个determineCurrentLookupKey()方法，得到指定的值。

**直观的想法，可能是定义一个静态变量，但是静态变量存在线程安全问题，所以我们需要为每一个线程都维护一个变量，用于指定连接的数据库是哪个，所以我们这里使用线程的副本ThreadLocal类**

#### 在com.bjpowernode.multi.dynamic包下创建ThreadLocalHolder类

public class ThreadLocalHolder {  
 public static final ThreadLocal<String> *holder* = new ThreadLocal<String>();  
 */\*\*  
 \* 向当前线程变量的副本中放一个数据源的key  
 \*/* public static void setDataSourceKey(String dataSourceKey){  
 *holder*.set(dataSourceKey);  
 }  
  
 */\*\*  
 \* 从当前线程变量的副本中取出放入数据源的key  
 \*/* public static String getDataSourceKey(){  
 return *holder*.get();  
 }  
}

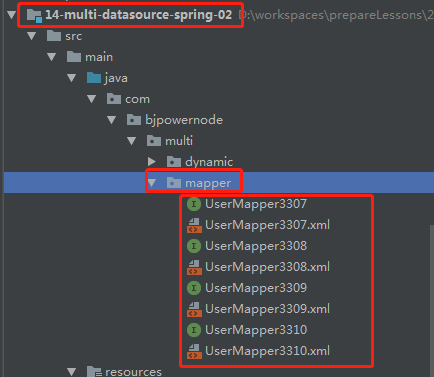
#### 对DynamicDataSource类中的方法进行实现

protected Object determineCurrentLookupKey() {  
 //在操作数据库前要确定是哪个数据源的key返回  
 return ThreadLocalHolder.*getDataSourceKey*();  
}

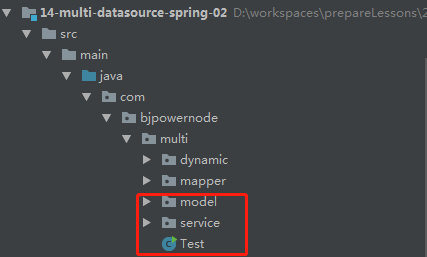
### 在applicationContext-mybatis.xml配置Mapper包扫描器，直接扫描mapper包即可

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <property name="sqlSessionFactoryBeanName" value="sessionFactory"/>  
 <property name="basePackage" value="com.bjpowernode.multi.mapper"/>  
</bean>

### 从14-multi-datasource-spring-01中拷贝一份UserMapper接口以及映射文件到14-multi-datasource-spring-02的mapper包下，将名字修改一下，并修改映射文件中的命名空间



### 将14-multi-datasource-spring-01中的model、service包以及Test类拷贝到将14-multi-datasource-spring-02项目中



### 修改UserServiceImpl中注入的UserMapper路径及包

@Autowired  
private com.bjpowernode.multi.mapper.UserMapper3307 userMapper3307;  
@Autowired  
private com.bjpowernode.multi.mapper.UserMapper3308 userMapper3308;  
@Autowired  
private com.bjpowernode.multi.mapper.UserMapper3309 userMapper3309;  
@Autowired  
private com.bjpowernode.multi.mapper.UserMapper3310 userMapper3310;

### 在applicationContext.xml文件中扫描impl包

<context:component-scan base-package="com.bjpowernode.multi.service.impl"/>

### 运行测试类，获取不同数据源用户

**注意：在操作数据库前，用指定数据源**

public class Test {  
 public static void main(String[] args) {  
 ClassPathXmlApplicationContext context =  
 new ClassPathXmlApplicationContext("classpath:applicationContext.xml");  
 UserService userService = context.getBean("userServiceImpl",UserService.class);  
 ThreadLocalHolder.*setDataSourceKey*("3307");  
 System.*out*.println("3307数据库：" +userService.getUserByIdFrom3307(4).getName());  
 ThreadLocalHolder.*setDataSourceKey*("3308");  
 System.*out*.println("3308数据库：" +userService.getUserByIdFrom3308(3).getName());  
 ThreadLocalHolder.*setDataSourceKey*("3309");  
 System.*out*.println("3309数据库：" +userService.getUserByIdFrom3309(4).getName());  
 ThreadLocalHolder.*setDataSourceKey*("3310");  
 System.*out*.println("3310数据库：" +userService.getUserByIdFrom3310(4).getName());  
 }  
}

## Spring+Mybatis方案二加事务管理步骤

### 在applicationContext-mybatis.xml文件中为数据源加事务管理器，并开启事务注解

如果只有一个事务管理器，那么开启事务注解的时候，transaction-manager属性可以省略

<!--事务管理器 对动态数据源进行管理-->  
 <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dynamicDataSource"/>  
 </bean>  
 <tx:annotation-driven/>

### 修改UserServiceImpl更新用户方法上加的事务注解

@Transactional  
public int updateUserFrom3307(User user) {  
 int updateRow = userMapper3307.updateByPrimaryKeySelective(user);  
 System.*out*.println(updateRow);  
 int a = 10/0;  
 return updateRow;  
}  
@Transactional  
public int updateUserFrom3308(User user) {  
 int updateRow = userMapper3308.updateByPrimaryKeySelective(user);  
 System.*out*.println(updateRow);  
 int a = 10/0;  
 return updateRow;  
}  
@Transactional  
public int updateUserFrom3309(User user) {  
 int updateRow = userMapper3309.updateByPrimaryKeySelective(user);  
 System.*out*.println(updateRow);  
 int a = 10/0;  
 return updateRow;  
}  
@Transactional  
public int updateUserFrom3310(User user) {  
 int updateRow = userMapper3310.updateByPrimaryKeySelective(user);  
 System.*out*.println(updateRow);  
 int a = 10/0;  
 return updateRow;  
}

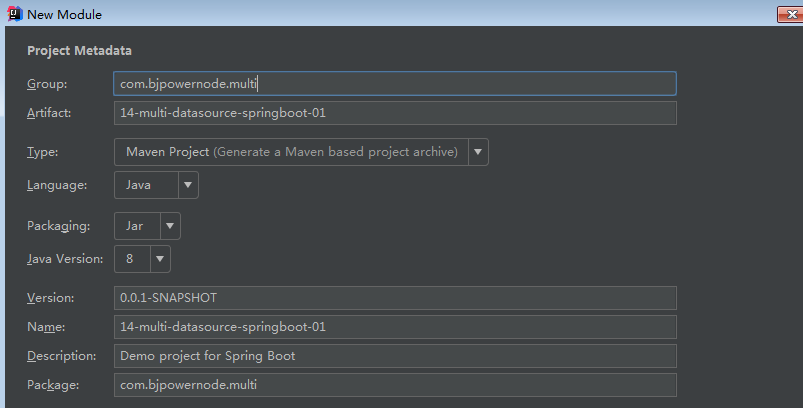
### 在Test类中进行测试

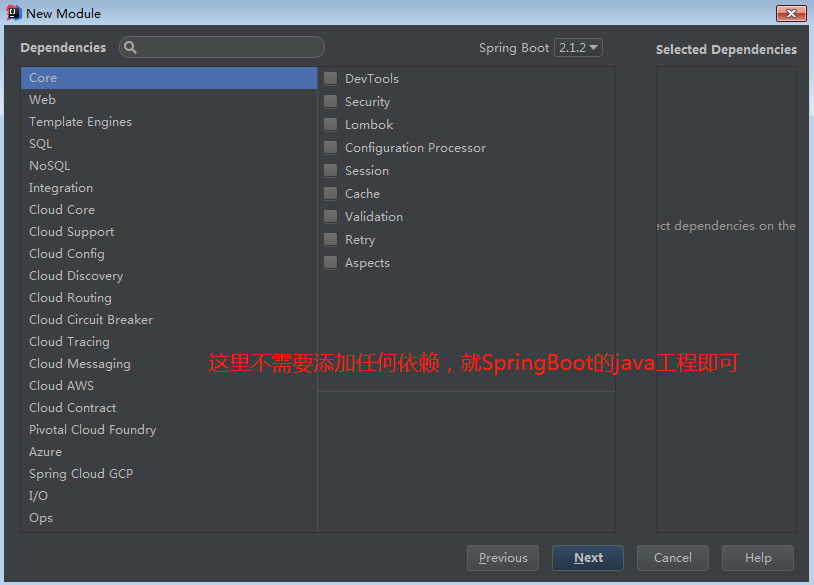
**进行数据库操作前，需要先指定数据源**

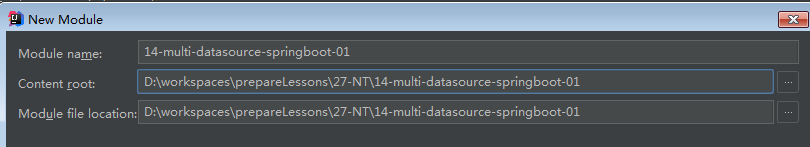
## SpringBoot+Mybatis方案一实现步骤

**核心思想：基于Mapper包的隔离，每个Mapper包操作不同的数据库，每个Mapper包对应一个数据库**

### 创建一个SpringBoot的java项目14-multi-datasource-springboot-01







### 在pom.xml文件中添加mybatis的依赖

<!-- 加载mybatis整合springboot -->  
<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <!--在springboot的父工程中没有指定版本，我们需要手动指定-->  
 <version>1.3.2</version>  
</dependency>  
<!-- MySQL的jdbc驱动包 -->  
<dependency>  
 <groupId>mysql</groupId>  
 <!--在springboot的父工程中指定了版本，我们就不需要手动指定了-->  
 <artifactId>mysql-connector-java</artifactId>  
</dependency>

<!--JDBC 数据库连接池-->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.1.1</version>  
</dependency>

### 在pom.xml文件中指定resource，对Mybatis映射文件编译

<resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>  
</resources>

### 在SpringBoot的核心配置文件中，配置连接信息

那么这里我们涉及多数据源，所以需要配置多个连接信息，如果配置多个，SpringBoot会报错，所以我们只能对使用自定义配置，然后在程序中获取

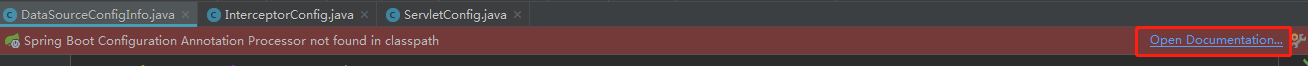
#3307数据库的连接配置信息  
spring.datasource.username3307=root  
spring.datasource.password3307=123456  
spring.datasource.driver3307=com.mysql.cj.jdbc.Driver  
spring.datasource.url3307=jdbc:mysql://192.168.235.128:3307/test?useUnicode=true&characterEncoding=utf8&useSSL=false  
#3308数据库的连接配置信息  
spring.datasource.username3308=root  
spring.datasource.password3308=123456  
spring.datasource.driver3308=com.mysql.cj.jdbc.Driver  
spring.datasource.url3308=jdbc:mysql://192.168.235.128:3308/test?useUnicode=true&characterEncoding=utf8&useSSL=false  
#3309数据库的连接配置信息  
spring.datasource.username3309=root  
spring.datasource.password3309=123456  
spring.datasource.driver3309=com.mysql.cj.jdbc.Driver  
spring.datasource.url3309=jdbc:mysql://192.168.235.128:3309/test?useUnicode=true&characterEncoding=utf8&useSSL=false  
#3310数据库的连接配置信息  
spring.datasource.username3310=root  
spring.datasource.password3310=123456  
spring.datasource.driver3310=com.mysql.cj.jdbc.Driver  
spring.datasource.url3310=jdbc:mysql://192.168.235.128:3310/test?useUnicode=true&characterEncoding=utf8&useSSL=false

### 在com.bjpowernode.multi.config包下定义一个映射核心文件中自定义信息的实体类DataSourceConfigInfo

@Component  
@ConfigurationProperties(prefix ="spring.datasource")  
public class DataSourceConfigInfo {  
 private String username3307;  
 private String password3307;  
 private String driver3307;  
 private String url3307;  
   
 private String username3308;  
 private String password3308;  
 private String driver3308;  
 private String url3308;  
   
 private String username3309;  
 private String password3309;  
 private String driver3309;  
 private String url3309;  
   
 private String username3310;  
 private String password3310;  
 private String driver3310;  
 private String url3310;

//省略get|set  
}

### 在自定义属性映射文件出现的警告处理



点击红色圈的链接，到SpringBoot官网上，拷贝如下依赖到pom.xml文件中，这个警告不处理也不会影响程序的运行

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-configuration-processor</artifactId>  
 <optional>true</optional>  
</dependency>

### 将Spring+Mybatis方案一实现中的applicationContext-mybatis.xml配置信息用SpringBoot注解替换

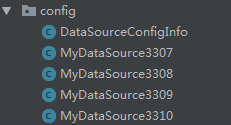
#### 因为我们这里有多个数据源，所有每个数据源对应一个配置类

#### 在com.bjpowernode.multi.config下创建MyDataSource3307类

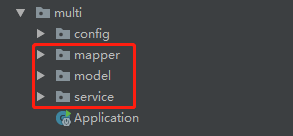
**注意：  
 在SpringBoot下，不能简单的配置MapperScannerConfigurer这个bean  
 在SpringBoot下，需要SqlSessionTemplate + @MapperScan注解 配合实现原来xml的功能**

@Configuration //相当于一个xml文件  
@MapperScan(basePackages="com.bjpowernode.multi.mapper.mapper3307"  
 ,sqlSessionTemplateRef = "sqlSessionTemplate3307")  
public class MyDataSource3307 {  
  
 @Autowired//注入我们自定义属性的映射类  
 private DataSourceConfigInfo dataSourceConfigInfo;  
  
 */\*\*  
 <bean id="dataSource3307" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3307/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
 </bean>  
 \*/* @Bean//相当于xml中的bean  
 public DruidDataSource dataSource3307(){  
 DruidDataSource druidDataSource = new DruidDataSource();  
 druidDataSource.setUrl(dataSourceConfigInfo.getUrl3307());  
 druidDataSource.setDriverClassName(dataSourceConfigInfo.getDriver3307());  
 druidDataSource.setUsername(dataSourceConfigInfo.getUsername3307());  
 druidDataSource.setPassword(dataSourceConfigInfo.getPassword3307());  
 return druidDataSource;  
 }  
  
 */\*\*  
 \** <!--3307数据库的sessionFactory-->  
 *<bean id="sessionFactory3307" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource3307"/>  
 </bean>  
 \*/* @Bean  
 public SqlSessionFactoryBean sessionFactory3307(){  
 SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();  
 sqlSessionFactoryBean.setDataSource(dataSource3307());  
 return sqlSessionFactoryBean;  
 }  
  
 */\*\*  
 \** <!--扫描3307库对应的mapper包，也就是说该Mapper下的接口操作的是3307数据库-->  
 *<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <property name="sqlSessionFactoryBeanName" value="sessionFactory3307"/>  
 <property name="basePackage" value="com.bjpowernode.multi.mapper.mapper3307"/>  
 </bean>  
 注意：  
 在SpringBoot下，不能简单的配置MapperScannerConfigurer这个bean  
 在SpringBoot下，需要SqlSessionTemplate + @MapperScan注解 配合实现原来xml的功能  
 \*/* @Bean  
 public SqlSessionTemplate sqlSessionTemplate3307() throws Exception{  
 //sessionFactory3307()方法返回SqlSessionFactoryBean，需要调用getObject返回SqlSessionFactory  
 SqlSessionTemplate sqlSessionTemplate =   
 new SqlSessionTemplate(sessionFactory3307().getObject());  
 return sqlSessionTemplate;  
 }  
}

#### 复制MyDataSource3307类为3308|3309|3310，并改变类中的内容为对应的数据源编号



### 从14-multi-datasource-spring-01中拷贝model，service到当前14-multi-datasource-springboot-01中



### 在Application中添加测试代码进行测试

@SpringBootApplication  
public class Application {  
 public static void main(String[] args) {  
 ConfigurableApplicationContext context = SpringApplication.*run*(Application.class, args);  
 UserService userService = context.getBean("userServiceImpl",UserService.class);  
 System.*out*.println("3307数据库：" +userService.getUserByIdFrom3307(4).getName());  
 System.*out*.println("3308数据库：" +userService.getUserByIdFrom3308(3).getName());  
 System.*out*.println("3309数据库：" +userService.getUserByIdFrom3309(4).getName());  
 System.*out*.println("3310数据库：" +userService.getUserByIdFrom3310(4).getName());  
 }  
}

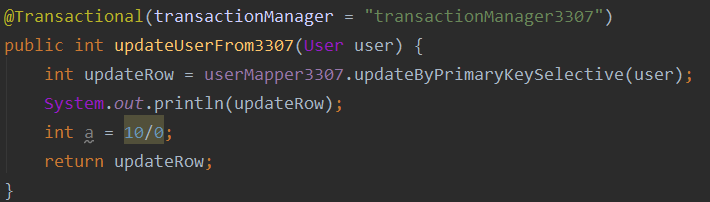
## SpringBoot+Mybatis方案一加事务管理步骤

### 在MyDataSource3307类中添加对事物的处理

*/\*\*  
 \* <bean id="transactionManager3307" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource3307"/>  
 </bean>  
 <tx:annotation-driven transaction-manager="transactionManager3307"/>  
 \*/*@Bean("transactionManager3307")  
public DataSourceTransactionManager transactionManager3307(){  
 DataSourceTransactionManager dataSourceTransactionManager   
 = new DataSourceTransactionManager();  
 dataSourceTransactionManager.setDataSource(dataSource3307());  
 return dataSourceTransactionManager;  
}

### 在MyDataSource3308|3309|3310类中同样添加对事物的处理

### 在UserServiceImpl实现类中需要控制事务的方法上添加事务注解，例如



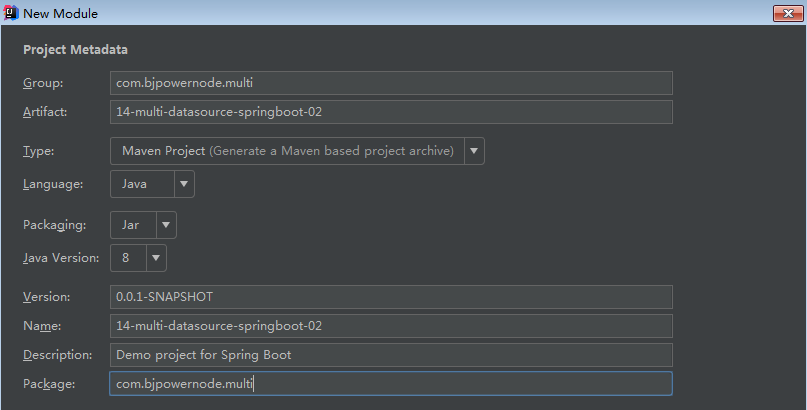
### 在Application类中编写代码测试

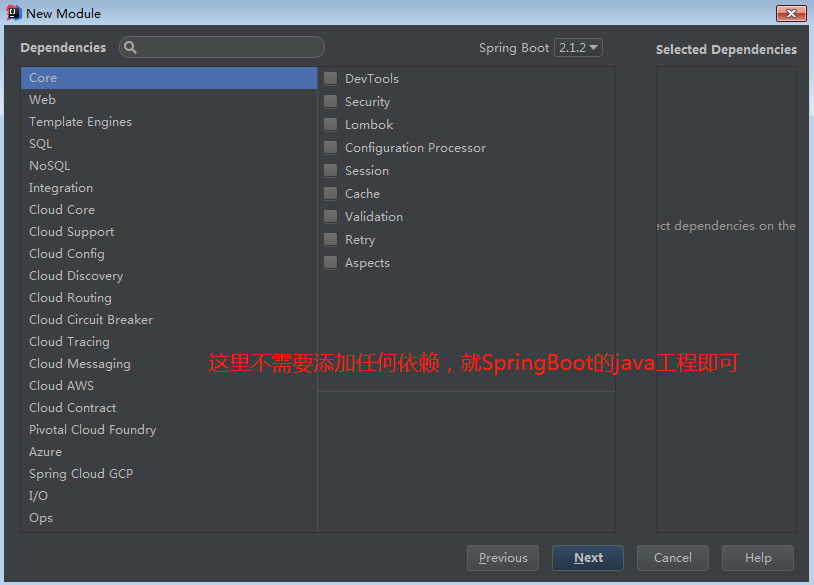
User user = new User();  
user.setId(3);  
user.setName("boot-3309123");  
userService.updateUserFrom3309(user);

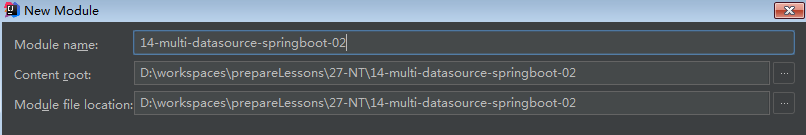
## SpringBoot+Mybatis方案二实现步骤

**核心思想：基于动态数据源，在运行的时候才知道要是用哪个数据源**

### 创建一个SpringBoot的java项目14-multi-datasource-springboot-02







### 在pom.xml文件中添加mybatis的依赖

<!-- 加载mybatis整合springboot -->  
<dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <!--在springboot的父工程中没有指定版本，我们需要手动指定-->  
 <version>1.3.2</version>  
</dependency>  
<!-- MySQL的jdbc驱动包 -->  
<dependency>  
 <groupId>mysql</groupId>  
 <!--在springboot的父工程中指定了版本，我们就不需要手动指定了-->  
 <artifactId>mysql-connector-java</artifactId>  
</dependency>

<!--JDBC 数据库连接池-->  
<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 <version>1.1.1</version>  
</dependency>

### 在pom.xml文件中指定resource，对Mybatis映射文件编译

<resources>  
 <resource>  
 <directory>src/main/java</directory>  
 <includes>  
 <include>\*\*/\*.xml</include>  
 </includes>  
 </resource>  
</resources>

### 在SpringBoot的核心配置文件中，配置连接信息

那么这里我们涉及多数据源，所以需要配置多个连接信息，如果配置多个，SpringBoot会报错，所以我们只能对使用自定义配置，然后在程序中获取

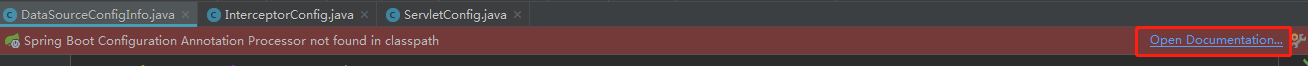
#3307数据库的连接配置信息  
spring.datasource.username3307=root  
spring.datasource.password3307=123456  
spring.datasource.driver3307=com.mysql.cj.jdbc.Driver  
spring.datasource.url3307=jdbc:mysql://192.168.235.128:3307/test?useUnicode=true&characterEncoding=utf8&useSSL=false  
#3308数据库的连接配置信息  
spring.datasource.username3308=root  
spring.datasource.password3308=123456  
spring.datasource.driver3308=com.mysql.cj.jdbc.Driver  
spring.datasource.url3308=jdbc:mysql://192.168.235.128:3308/test?useUnicode=true&characterEncoding=utf8&useSSL=false  
#3309数据库的连接配置信息  
spring.datasource.username3309=root  
spring.datasource.password3309=123456  
spring.datasource.driver3309=com.mysql.cj.jdbc.Driver  
spring.datasource.url3309=jdbc:mysql://192.168.235.128:3309/test?useUnicode=true&characterEncoding=utf8&useSSL=false  
#3310数据库的连接配置信息  
spring.datasource.username3310=root  
spring.datasource.password3310=123456  
spring.datasource.driver3310=com.mysql.cj.jdbc.Driver  
spring.datasource.url3310=jdbc:mysql://192.168.235.128:3310/test?useUnicode=true&characterEncoding=utf8&useSSL=false

### 在com.bjpowernode.multi.config包下定义一个映射核心文件中自定义信息的实体类DataSourceConfigInfo

@Component  
@ConfigurationProperties(prefix ="spring.datasource")  
public class DataSourceConfigInfo {  
 private String username3307;  
 private String password3307;  
 private String driver3307;  
 private String url3307;  
   
 private String username3308;  
 private String password3308;  
 private String driver3308;  
 private String url3308;  
   
 private String username3309;  
 private String password3309;  
 private String driver3309;  
 private String url3309;  
   
 private String username3310;  
 private String password3310;  
 private String driver3310;  
 private String url3310;

//省略get|set  
}

### 在自定义属性映射文件出现的警告处理

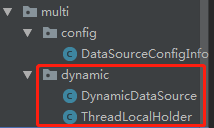


点击红色圈的链接，到SpringBoot官网上，拷贝如下依赖到pom.xml文件中，这个警告不处理也不会影响程序的运行

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-configuration-processor</artifactId>  
 <optional>true</optional>  
</dependency>

### 将Spring+Mybatis方案二实现中的applicationContext-mybatis.xml配置信息用SpringBoot注解替换

#### 因为方案二是基于动态数据源的，所以先将我们前面14-multi-datasource-spring-02中定义的动态数据源DynamicDataSource类以及ThreadLocal类拷贝到14-multi-datasource-springboot-02中

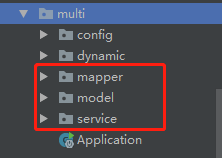


#### 在com.bjpowernode.multi.config下创建MyDataSource类

@Configuration //相当于一个xml文件  
@MapperScan(basePackages="com.bjpowernode.multi.mapper",sqlSessionTemplateRef = "sqlSessionTemplate")  
public class MyDataSource {  
 @Autowired//注入我们自定义属性的映射类  
 private DataSourceConfigInfo dataSourceConfigInfo;  
  
 */\*\*  
 <bean id="dataSource3307" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3307/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
 </bean>  
 \*/* @Bean//相当于xml中的bean  
 public DruidDataSource dataSource3307(){  
 DruidDataSource druidDataSource = new DruidDataSource();  
 druidDataSource.setUrl(dataSourceConfigInfo.getUrl3307());  
 druidDataSource.setDriverClassName(dataSourceConfigInfo.getDriver3307());  
 druidDataSource.setUsername(dataSourceConfigInfo.getUsername3307());  
 druidDataSource.setPassword(dataSourceConfigInfo.getPassword3307());  
 return druidDataSource;  
 }  
 */\*\*  
 <bean id="dataSource3308" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3308/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
 </bean>  
 \*/* @Bean//相当于xml中的bean  
 public DruidDataSource dataSource3308(){  
 DruidDataSource druidDataSource = new DruidDataSource();  
 druidDataSource.setUrl(dataSourceConfigInfo.getUrl3308());  
 druidDataSource.setDriverClassName(dataSourceConfigInfo.getDriver3308());  
 druidDataSource.setUsername(dataSourceConfigInfo.getUsername3308());  
 druidDataSource.setPassword(dataSourceConfigInfo.getPassword3308());  
 return druidDataSource;  
 }  
 */\*\*  
 <bean id="dataSource3309" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3309/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
 </bean>  
 \*/* @Bean//相当于xml中的bean  
 public DruidDataSource dataSource3309(){  
 DruidDataSource druidDataSource = new DruidDataSource();  
 druidDataSource.setUrl(dataSourceConfigInfo.getUrl3309());  
 druidDataSource.setDriverClassName(dataSourceConfigInfo.getDriver3309());  
 druidDataSource.setUsername(dataSourceConfigInfo.getUsername3309());  
 druidDataSource.setPassword(dataSourceConfigInfo.getPassword3309());  
 return druidDataSource;  
 }  
 */\*\*  
 <bean id="dataSource3310" class="com.alibaba.druid.pool.DruidDataSource">  
 <property name="url" value="jdbc:mysql://192.168.235.128:3310/test"/>  
 <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  
 <property name="username" value="root"/>  
 <property name="password" value="123456"/>  
 </bean>  
 \*/* @Bean//相当于xml中的bean  
 public DruidDataSource dataSource3310(){  
 DruidDataSource druidDataSource = new DruidDataSource();  
 druidDataSource.setUrl(dataSourceConfigInfo.getUrl3310());  
 druidDataSource.setDriverClassName(dataSourceConfigInfo.getDriver3310());  
 druidDataSource.setUsername(dataSourceConfigInfo.getUsername3310());  
 druidDataSource.setPassword(dataSourceConfigInfo.getPassword3310());  
 return druidDataSource;  
 }  
 */\*\*  
 \** <!--数据库的sessionFactory-->  
 *<bean id="sessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dynamicDataSource"/>  
 </bean>  
 \*/* @Bean  
 public SqlSessionFactoryBean sessionFactory(){  
 SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();  
 sqlSessionFactoryBean.setDataSource(dynamicDataSource());  
 return sqlSessionFactoryBean;  
 }  
 */\*\*  
 \** <!--动态数据源-->  
 *<bean id="dynamicDataSource" class="com.bjpowernode.multi.dynamic.DynamicDataSource">  
 <property name="targetDataSources">  
 <map>  
 <entry key="3307" value-ref="dataSource3307"/>  
 <entry key="3308" value-ref="dataSource3308"/>  
 <entry key="3309" value-ref="dataSource3309"/>  
 <entry key="3310" value-ref="dataSource3310"/>  
 </map>  
 </property>  
 </bean>  
 \*/*

@Beanpublic DynamicDataSource dynamicDataSource(){  
 DynamicDataSource dynamicDataSource = new DynamicDataSource();  
 Map<Object, Object> targetDataSources = new ConcurrentHashMap<Object,Object>();  
 targetDataSources.put(DynamicDataSource.*DATASOURCE\_KEY\_3307*,dataSource3307());  
 targetDataSources.put(DynamicDataSource.*DATASOURCE\_KEY\_3308*,dataSource3308());  
 targetDataSources.put(DynamicDataSource.*DATASOURCE\_KEY\_3309*,dataSource3309());  
 targetDataSources.put(DynamicDataSource.*DATASOURCE\_KEY\_3310*,dataSource3310());  
 dynamicDataSource.setTargetDataSources(targetDataSources);  
 return dynamicDataSource;  
 }  
 */\*\*  
 \* <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <property name="sqlSessionFactoryBeanName" value="sessionFactory"/>  
 <property name="basePackage" value="com.bjpowernode.multi.mapper"/>  
 </bean>  
 注意：  
 在SpringBoot下，不能简单的配置MapperScannerConfigurer这个bean  
 在SpringBoot下，需要SqlSessionTemplate + @MapperScan注解 配合实现原来xml的功能  
 \*/* @Bean  
 public SqlSessionTemplate sqlSessionTemplate() throws Exception{  
 SqlSessionTemplate sqlSessionTemplate =new SqlSessionTemplate(sessionFactory().getObject());  
 return sqlSessionTemplate;  
 }  
}

### 将14-multi-datasource-spring-02中的mapper、service、model拷贝到14-multi-datasource-springboot-02中



### 在Application添加测试代码进行测试

**注意：操作数据库前需要指定数据源**

@SpringBootApplication  
public class Application {  
 public static void main(String[] args) {  
 ConfigurableApplicationContext context = SpringApplication.*run*(Application.class, args);  
 UserService userService = context.getBean("userServiceImpl",UserService.class);  
 ThreadLocalHolder.*setDataSourceKey*(DynamicDataSource.*DATASOURCE\_KEY\_3307*);  
 System.*out*.println("3307数据库：" +userService.getUserByIdFrom3307(4).getName());  
 ThreadLocalHolder.*setDataSourceKey*(DynamicDataSource.*DATASOURCE\_KEY\_3308*);  
 System.*out*.println("3308数据库：" +userService.getUserByIdFrom3308(3).getName());  
 ThreadLocalHolder.*setDataSourceKey*(DynamicDataSource.*DATASOURCE\_KEY\_3309*);  
 System.*out*.println("3309数据库：" +userService.getUserByIdFrom3309(4).getName());  
 ThreadLocalHolder.*setDataSourceKey*(DynamicDataSource.*DATASOURCE\_KEY\_3310*);  
 System.*out*.println("3310数据库：" +userService.getUserByIdFrom3310(4).getName());  
 }  
}

## SpringBoot+Mybatis方案二加事务管理步骤

### 在MyDataSource类中添加如下配置

*/\*\*  
 \** <!--事务管理器 对动态数据源进行管理-->  
 *<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dynamicDataSource"/>  
 </bean>  
 <tx:annotation-driven/>  
 \*/*@Bean  
public DataSourceTransactionManager transactionManager(){  
 DataSourceTransactionManager dataSourceTransactionManager = new DataSourceTransactionManager();  
 dataSourceTransactionManager.setDataSource(dynamicDataSource());  
 return dataSourceTransactionManager;  
}

### 在Application类上添加事务注解驱动，并测试

**注意：操作数据库前需要指定数据源**

@SpringBootApplication  
@EnableTransactionManagement  
public class Application {  
 public static void main(String[] args) {  
 ConfigurableApplicationContext context = SpringApplication.*run*(Application.class, args);  
 UserService userService = context.getBean("userServiceImpl",UserService.class);  
 User user = new User();  
 user.setId(3);  
 user.setName("boot-123");  
 ThreadLocalHolder.*setDataSourceKey*(DynamicDataSource.*DATASOURCE\_KEY\_3309*);  
 userService.updateUserFrom3309(user);  
 }  
}