MySQL 主从复制讲义

北京动力节点教育科技有限公司

动力节点课程讲义

DONGLIJIEDIANKECHENGJIANGYI

www.bjpowernode.com

第1章 MySQL 主从复制概述

1.1 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.

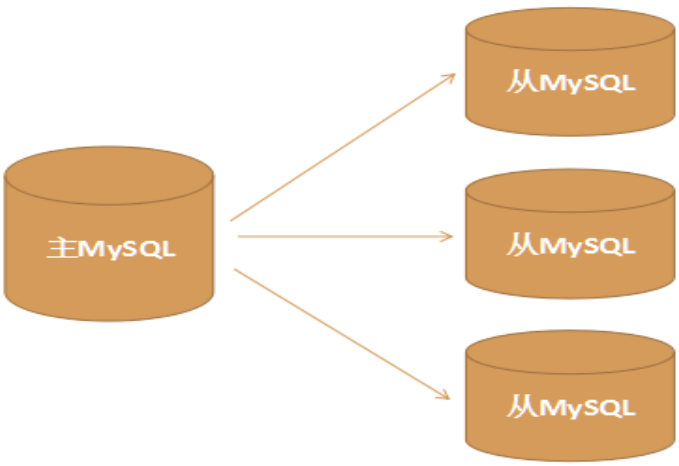
MySQL 提供了主从复制功能以提高服务的可用性与数据的安全可靠性.

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。

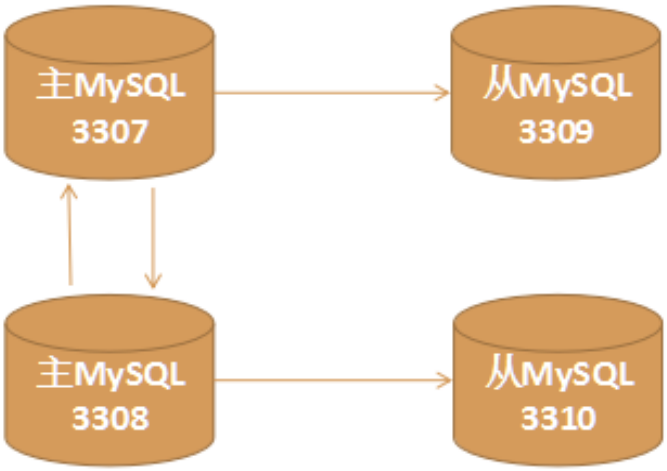
主从复制可以实现对数据库备份和读写分离

1.2 主从复制架构

1.2.1 一主多从架构



1.2.2 双主双从架构

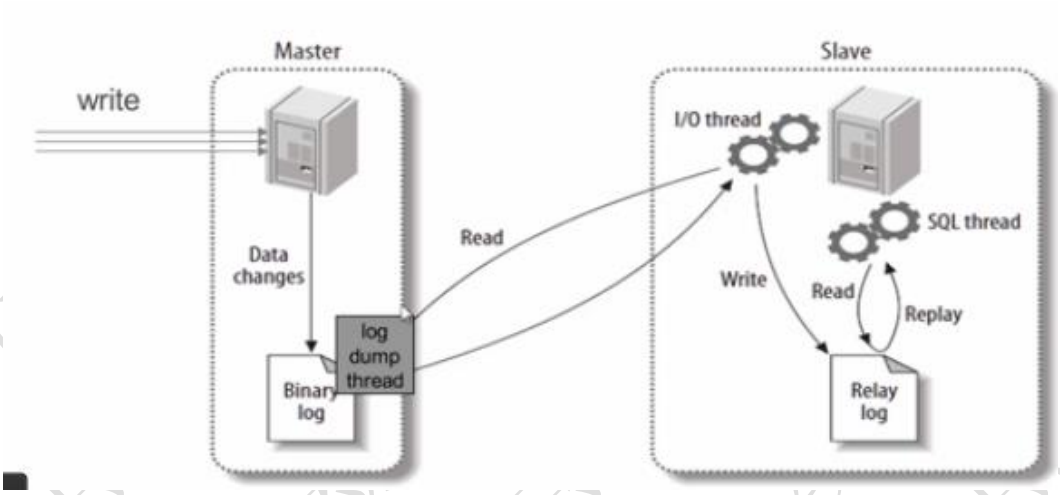


1.3 主从复制原理（流程|步骤）

- 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

1.4 MySQL 主从复制流程图



第2章 MySQL 多实例搭建

2.1 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

2.2 多实例配置

2.2.1 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

```
[root@localhost bin]# cd /usr/local/mysql-5.7.24/data/
[root@localhost data]# mkdir 3307
[root@localhost data]# mkdir 3308
[root@localhost data]# mkdir 3309
[root@localhost data]# mkdir 3310
[root@localhost data]# ll
total 188512
drwxr-xr-x. 2 root  root         6 Jan 31 11:01 3307
drwxr-xr-x. 2 root  root         6 Jan 31 11:01 3308
drwxr-xr-x. 2 root  root         6 Jan 31 11:01 3309
drwxr-xr-x. 2 root  root         6 Jan 31 11:01 3310
```

2.2.2 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

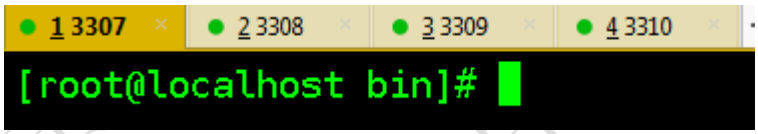
./mysqld --initialize-insecure --basedir=/usr/java/mysql --datadir=/usr/java/mysql/data/3307 --user=mysql

```
./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

2.2.3 配置四个 MySQL 数据库服务的 my.cnf 文件

（1） 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



（2） 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
```

2.3 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

2.4 启动后配置

2.4.1 登录进入每一个 MySQL 实例

在 mysql-5.7.24/bin 目录下执行命令

（1） 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

（2） 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

2.4.2 修改 mysql 的密码

alter user 'root'@'localhost' identified by '123456';
其中 123456 是我们设置的密码

2.4.3 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

2.4.4 刷新权限

flush privileges;

2.5 多实例关闭

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

（1） 方式一：使用端口、主机关闭 （推荐）

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

（2） 方式二:使用 socket 文件

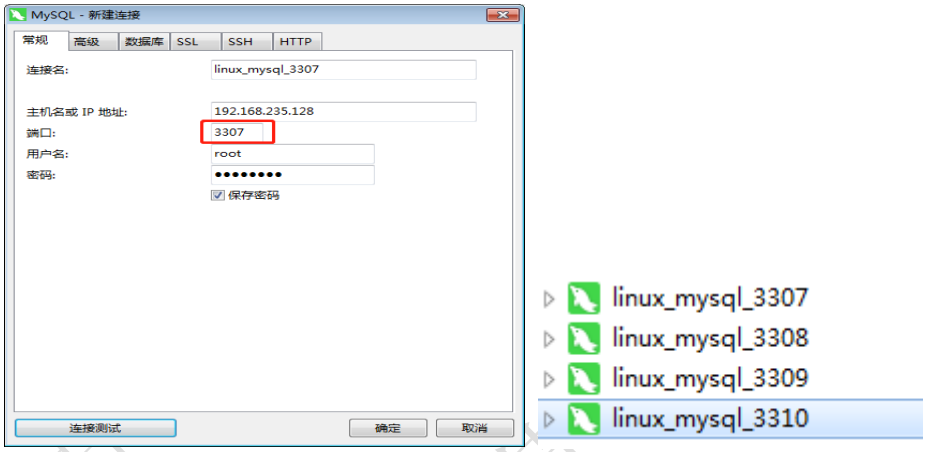
./mysqladmin -uroot -p -S /usr/local/mysql-5.7.24/data/3307/mysql.sock shutdown

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

（4） 退出 MySQL 命令行

在客户端命令行下执行 exit

2.6 在 Navicat 中创建多个连接，分别连接主从库

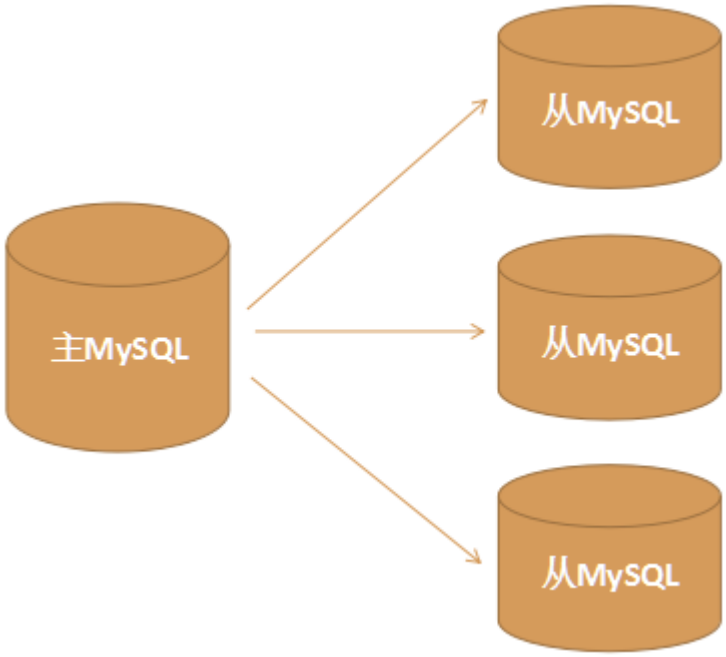


第3章 一主多从环境搭建

3.1 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

3.2 一主三从架构架构图



3.3 环境配置（先将各个服务器停掉,删除掉中文注释）

3.3.1 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
```

3.3.2 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
```

3.3.3 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
```

3.3.4 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
```

3.4 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 &

3.5 主从设置（重要）

3.5.1 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

（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

（2） 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

（3） 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;

```
mysql> show master status;
+------------------+----------+
| File             | Position |
+------------------+----------+
| mysql-bin.000001 |      154 |
+------------------+----------+
1 row in set (0.00 sec)
```

3.5.2 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

（1） 查看从服务器状态

show slave status;
初始状态：Empty set

（2） 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;

（3） 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;
```

（4） 在从机器上执行开始复制命令

start slave;

（5） 至此一个一主多从搭建完成

3.6 主从验证

3.6.1 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

3.6.2 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

3.6.3 查看主从复制 binlog 日志文件内容

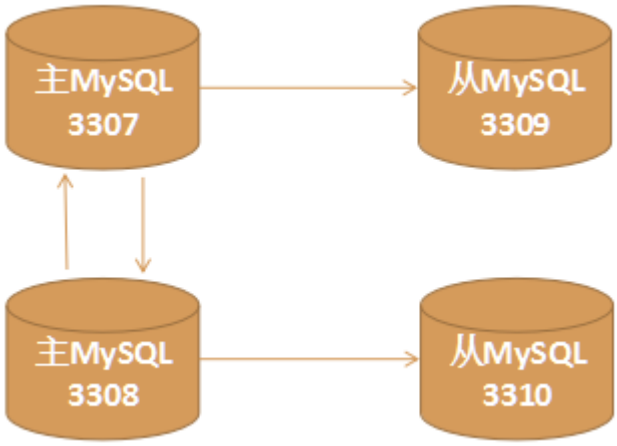
在主服务器客户端执行：show binlog events in 'mysql-bin.000001'\G;

第4章 双主双从环境搭建

4.1 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。

4.2 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

4.3 环境配置（不要含有中文注释）

4.3.1 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
```

4.3.2 在 MySQL 主服务器(3308)配置文件 my.cnf 里面加入

```
log-bin=mysql-bin
server-id=3308
```

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

```
server-id=3309
```

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

```
server-id=3310
```

4.3.5 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
```

4.3.6 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
```

4.3.7 配置项说明

（1） 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

（2） 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

（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

（4） 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。

（5） 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

4.4 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 &
```

4.5 主从设置（重要）

4.5.1 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

（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';
```

（2） 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

（3） 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;

```
mysql> show master status;
+------------------+----------+
| File             | Position |
+------------------+----------+
| mysql-bin.000001 |      154 |
+------------------+----------+
1 row in set (0.00 sec)
```

（4） 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;

4.5.2 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

（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;

（2） 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

A、 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;
```

B、 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;
```

（3） 在从机器上执行开始复制命令(4 台 MySQL 上都执行)

start slave;

（4） 至此双主双从就搭建好了

4.6 双主双从验证

4.6.1 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

4.6.2 在主服务器上操作测试数据的复制和同步情况

第5章 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

5.1 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

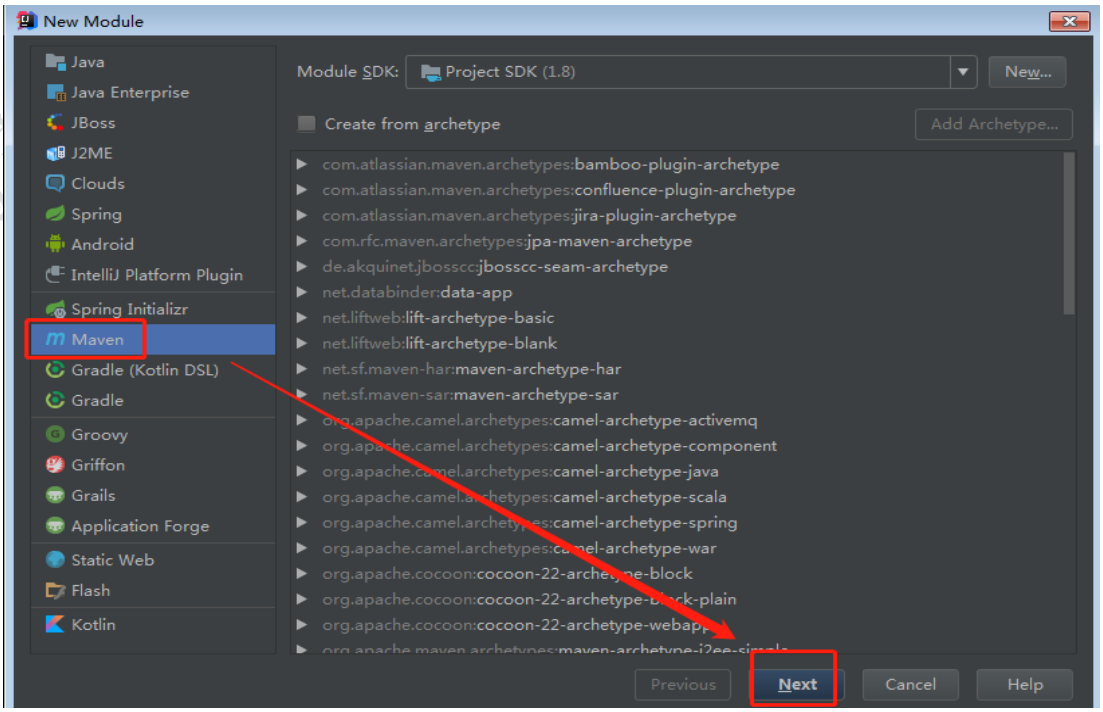
Springboot + JPA

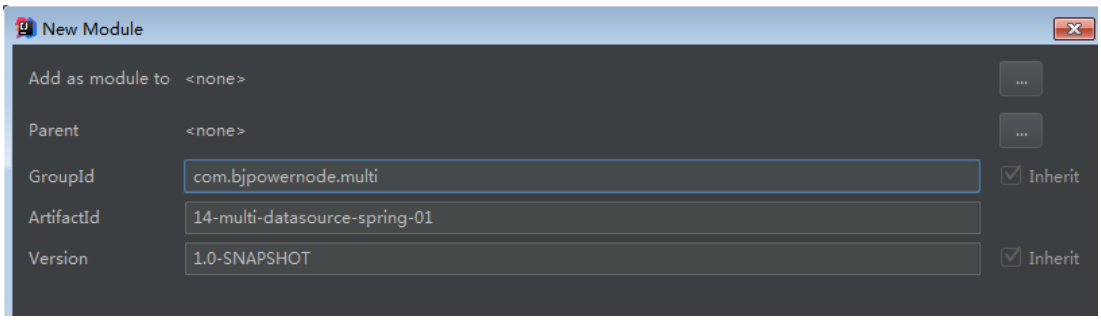
我们这里主要给大家介绍 Spring+Mybatis 和 SpringBoot+Mybatis 开发模式下的多数据源问题

5.2 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

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





5.2.2 在 pom.xml 文件中添加相关的依赖

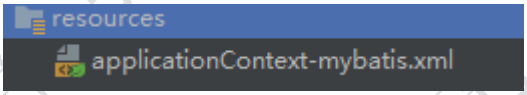
```
<!--Spring 相关的依赖-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.1.4.RELEASE</version>
</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>
```

5.2.3 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>
```

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



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

5.2.6 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>
```

5.2.7 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>
```

5.2.8 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

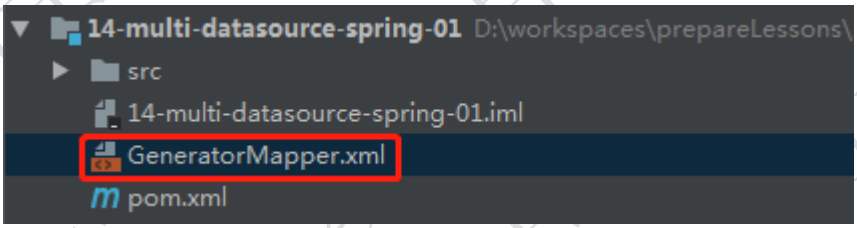
```
<!--扫描 3307 库对应的 mapper 包，也就是说该 Mapper 下的接口操作的是 3307 数据库-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="sqlSessionFactoryBeanName" value="sessionFactory3307"/>
    <property name="basePackage" value="com.bjpowernode.multi.mapper.mapper3307"/>
</bean>
<!--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

5.2.9 在 pom.xml 文件中加入 Mybatis 反向工程插件

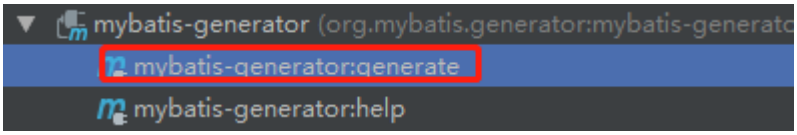
```
<!--mybatis 代码自动生成插件-->
<plugin>
    <groupId>org.mybatis.generator</groupId>
    <artifactId>mybatis-generator-maven-plugin</artifactId>
    <version>1.3.7</version>
    <configuration>
```

```
        <!--配置文件的位置-->
        <configurationFile>GeneratorMapper.xml</configurationFile>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
    </configuration>
</plugin>
```

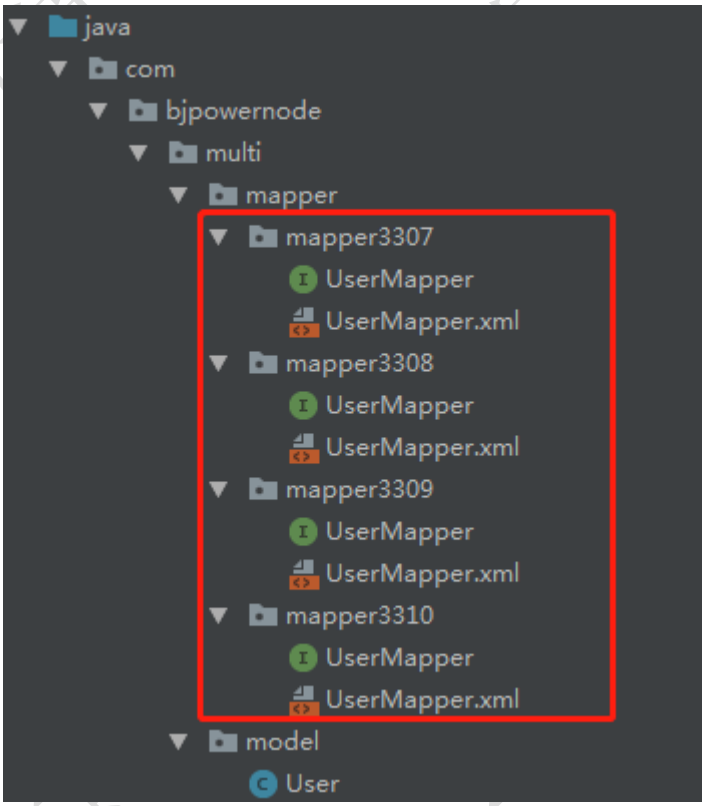
5.2.10 将反向工程的配置文件拷贝到当前项目中



5.2.11 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



5.2.12 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



5.2.13 修改子包下的 xml 文件的命名空间为对应的包

```
<mapper namespace="com.bjpowernode.multi.mapper.mapper3307.UserMapper">
```

5.2.14 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);
}
```

5.2.15 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);
    }
}
```

5.2.16 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

```
@Component("userMapper3307")
public interface UserMapper {
```

5.2.17 在 applicationContext.xml 文件中扫描 impl 包

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

5.2.18 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());
    }
}
```

5.3 Spring+Mybatis 方案一加事务管理步骤

5.3.1 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"/>
```

5.3.2 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);
```

5.3.3 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;
}
```

5.3.4 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

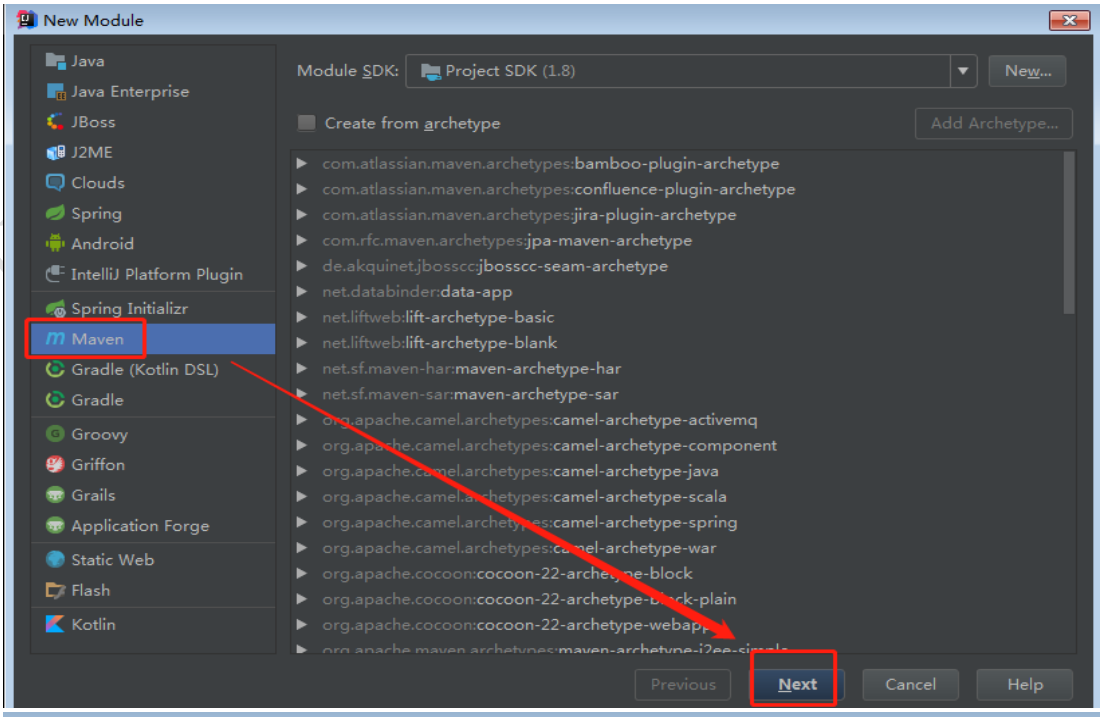
  //@Transactional(transactionManager = "transactionManager3309")

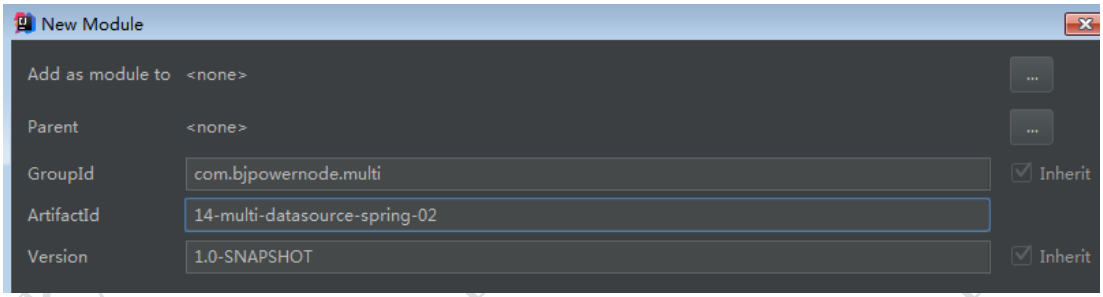
5.4 Spring+Mybatis 方案二实现步骤

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

5.4.1 创建一个普通的 maven 项目

14-multi-datasource-spring-02





5.4.2 在 pom.xml 文件中添加相关的依赖

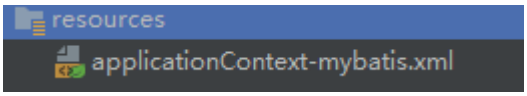
```
<!--Spring 相关的依赖-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.1.4.RELEASE</version>
</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>
```

5.4.3 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>
```

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



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

5.4.6 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>
```

5.4.7 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

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

（1） 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

（2） 该类中有个抽象方法 determineCurrentLookupKey 需要实现

```
protected Object determineCurrentLookupKey() {
    return null;
}
```

（3） 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";
```

（4） 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>
```

（5） 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

（6） 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();
```

```
    }
}
```

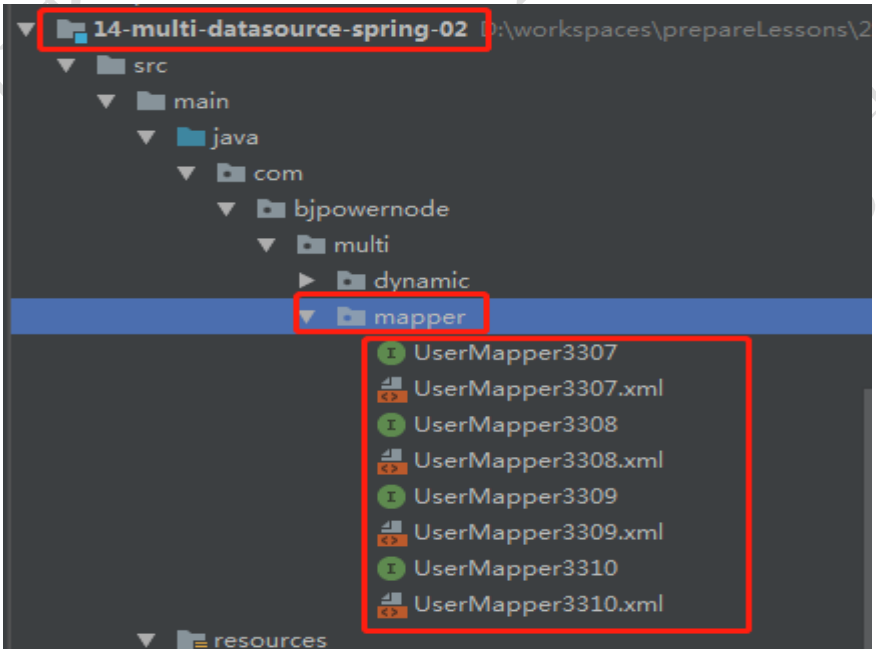
（7） 对 DynamicDataSource 类中的方法进行实现

```
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();
}
```

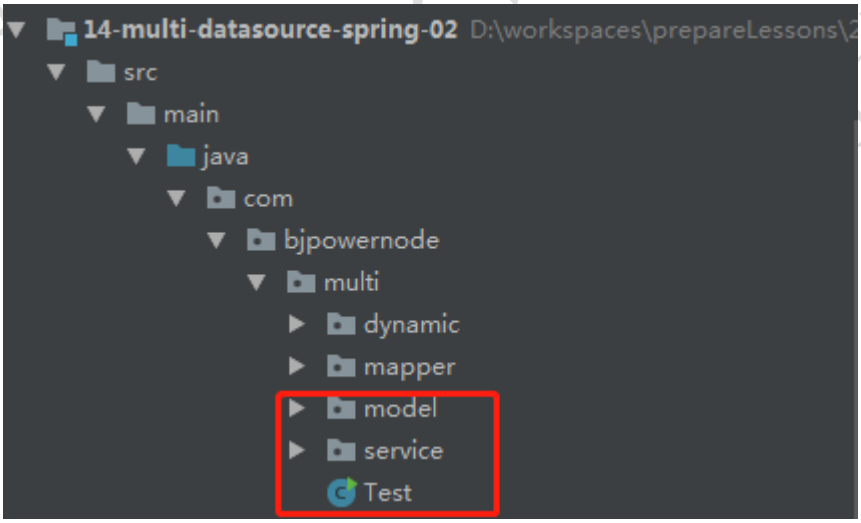
5.4.9 在 applicationContext-mybatis.xml 配置 Mapper 包扫描器，直接扫描 mapper 包即可

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

5.4.10 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



5.4.11 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



5.4.12 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;
```

5.4.13 在 applicationContext.xml 文件中扫描 impl 包

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

5.4.14 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());
    }
}
```

5.5 Spring+Mybatis 方案二加事务管理步骤

5.5.1 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/>
```

5.5.2 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;
}
```

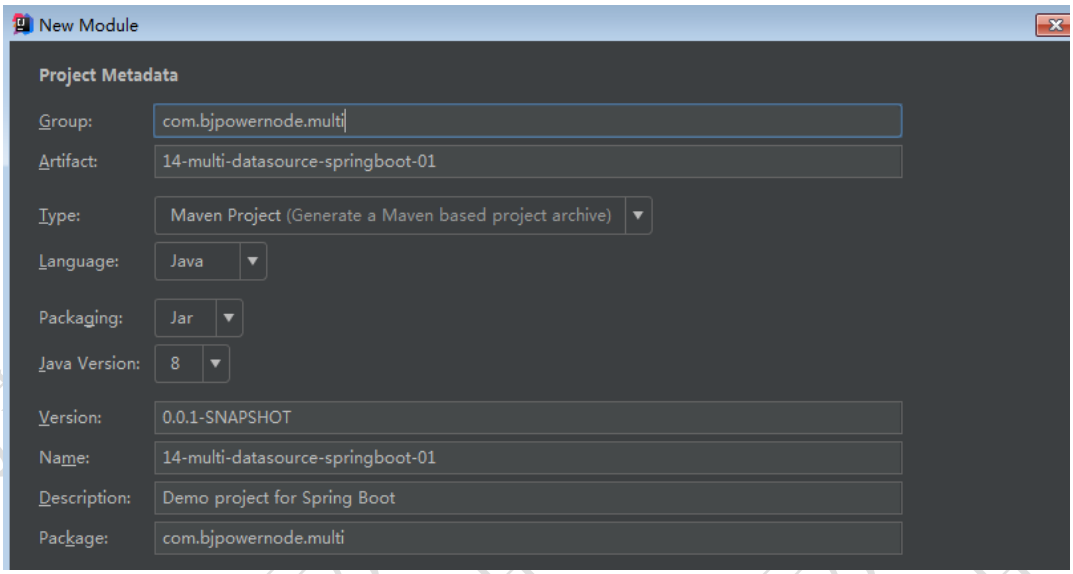
5.5.3 在 Test 类中进行测试

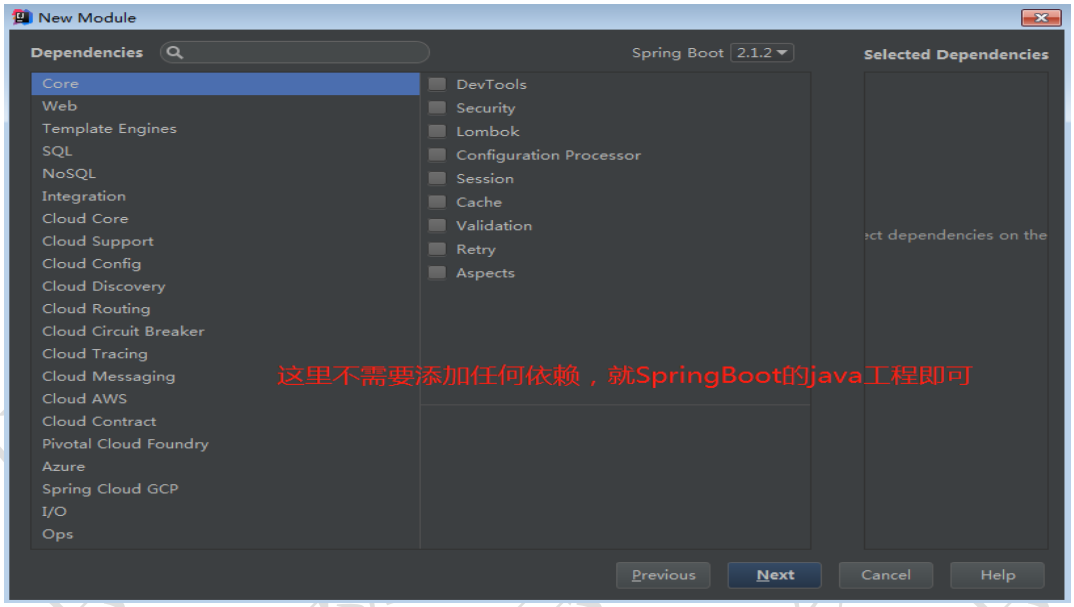
进行数据库操作前，需要先指定数据源

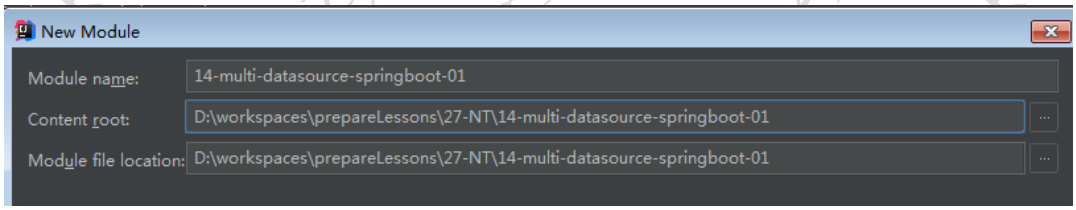
5.6 SpringBoot+Mybatis 方案一实现步骤

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

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







5.6.2 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>
```

5.6.3 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>
```

5.6.4 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?use
Unicode=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?use
Unicode=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?use
Unicode=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?use
Unicode=true&characterEncoding=utf8&useSSL=false
```

5.6.5 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
}
```

5.6.6 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>
```

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

（1） 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

（2） 在 com.bjpowernode.multi.config 下创建 MyDataSource3307 类

注意：

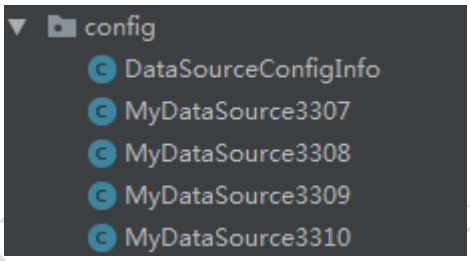
在 SpringBoot 下，不能简单的配置 MapperScannerConfigurer 这个 bean

在 SpringBoot 下，需要 SqlSessionTemplate + @MapperScan 注解 配合实现原来 xml 的功能

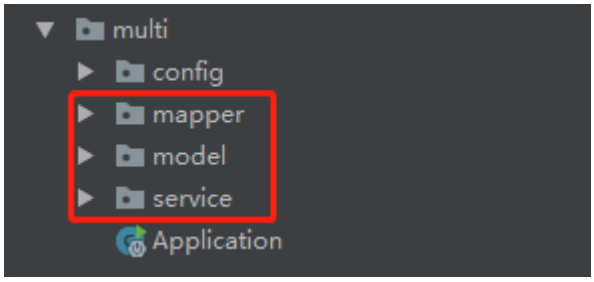
```
@Configuration //相当于一个 xml 文件
@MapperScan(basePackages="com.bjpowernode.multi.mapper.mapper3307"
        ,sqlSessionTemplateRef = "sqlSessionTemplate3307")
public class MyDataSource3307 {

    @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;
    }
}
```

（3） 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



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



5.6.9 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());
    }
}
```

5.7 SpringBoot+Mybatis 方案一加事务管理步骤

5.7.1 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;
}
```

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

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

```
@Transactional(transactionManager = "transactionManager3307")
public int updateUserFrom3307(User user) {
    int updateRow = userMapper3307.updateByPrimaryKeySelective(user);
    System.out.println(updateRow);
    int a = 10/0;
    return updateRow;
}
```

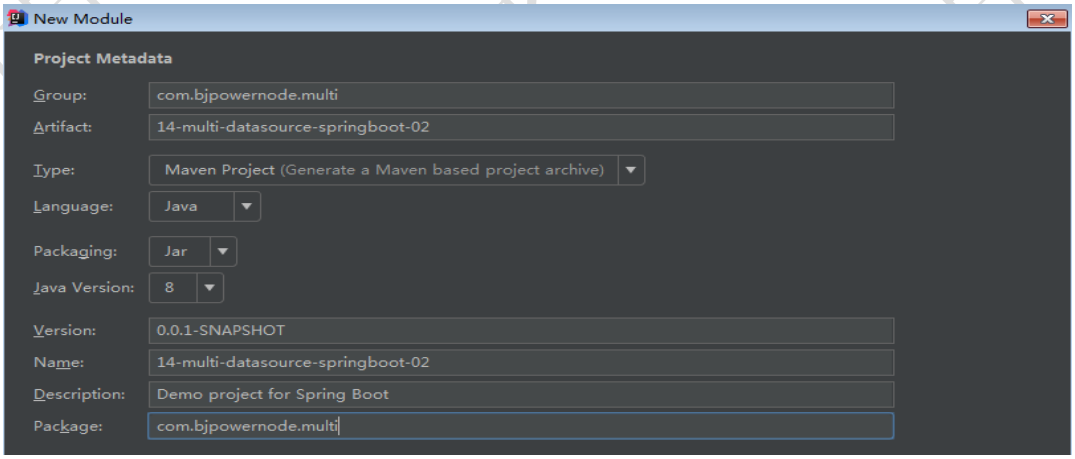
5.7.4 在 Application 类中编写代码测试

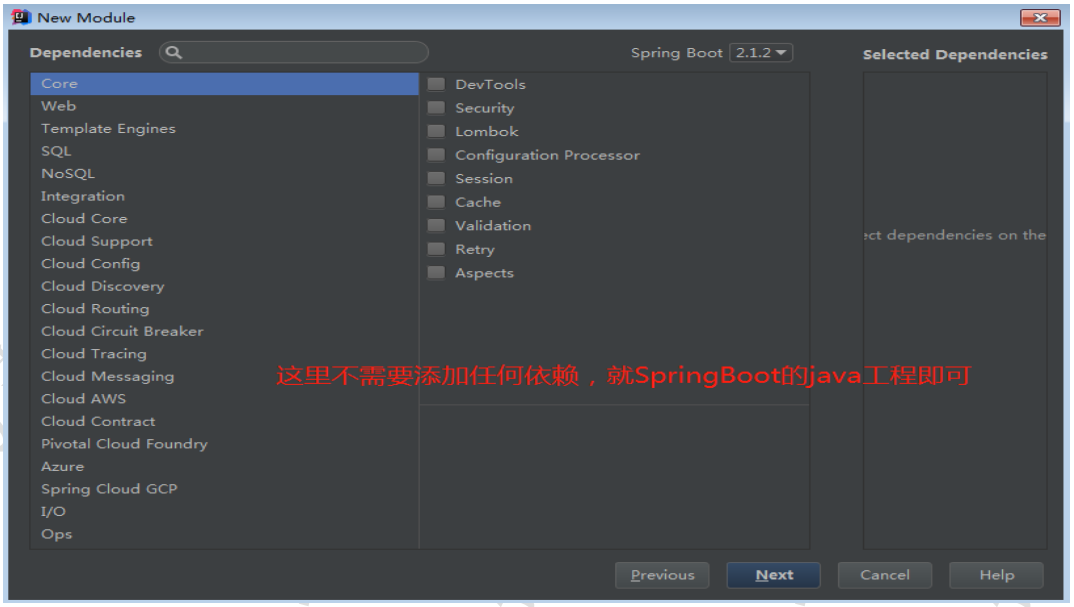
```
User user = new User();
user.setId(3);
user.setName("boot-3309123");
userService.updateUserFrom3309(user);
```

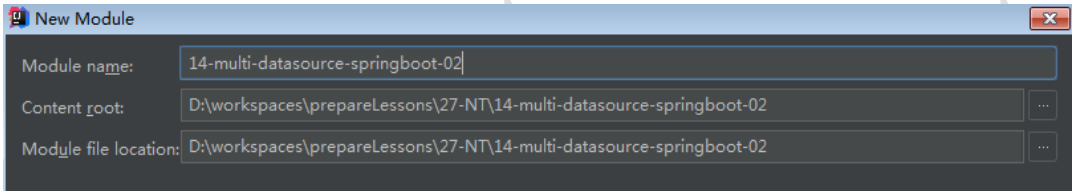
5.8 SpringBoot+Mybatis 方案二实现步骤

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

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







5.8.2 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>
```

5.8.3 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>
```

5.8.4 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?use
Unicode=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?use
Unicode=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?use
Unicode=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?use
Unicode=true&characterEncoding=utf8&useSSL=false
```

5.8.5 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
}
```

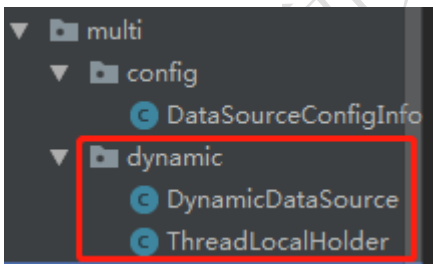
5.8.6 在自定义属性映射文件出现的警告处理

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>
```

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

（1）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



（2） 在 com.bjpowernode.multi.config 下创建 MyDataSource 类

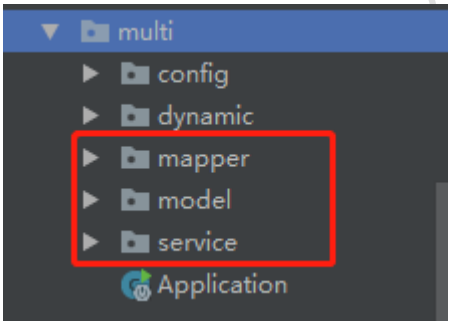
```
@Configuration //相当于一个 xml 文件
@MapperScan(basePackages="com.bjpowernode.multi.mapper",sqlSessionTemplateRef =
"sqlSessionTemplate")
public class MyDataSource {
    @Autowired//注入我们自定义属性的映射类
    private DataSourceConfigInfo dataSourceConfigInfo;

    /**
     <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>
     */
    @Bean
    public 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;
    }
}
```

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



5.8.9 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());
   }
}
```

5.9 SpringBoot+Mybatis 方案二加事务管理步骤

5.9.1 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.DataSourceTransactionMana
ger">
        <property name="dataSource" ref="dynamicDataSource"/>
    </bean>
     <tx:annotation-driven/>
 */
@Bean
public DataSourceTransactionManager transactionManager(){
    DataSourceTransactionManager dataSourceTransactionManager = new
DataSourceTransactionManager();
    dataSourceTransactionManager.setDataSource(dynamicDataSource());
    return dataSourceTransactionManager;
}
```

5.9.2 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);
   }
}
```