SpringBoot2从零开始（二）——多数据源配置 [原创]

SpringBoot (list?tag=SpringBoot) 多数据源 (list?tag=多数据源) MyBatis (list?tag=MyBatis)

wjyuian (mailto:wjyuian@126.com) 2018-09-19 19:44:25 2584

零、前言

多数据源配置，一直是大部分企业级开发所要面对的，即使微服务化概念提出来已经有一点时间了。微服务边界定义的好坏，直接影响系统的真实复杂度。

所以即使用了提倡微服务的 SpringBoot ，即使 SpringBoot 默认约定配置是单数据源的，多数据源整合还是有其存在的必要的。

当然，直接将非 SpringBoot 项目中多数据源配置的配置文件挪过来，稍作修改亦可实现。但是，既然使用了 SpringBoot ，如果依旧用那种方式，如何体现使用 SpringBoot 的“优越感”？至少要通过使用 SpringBoot 提供的诸多便利方式来实现多数据源，方可自称是 SpringBoot 的 脑残粉 ，啊不，是 倡导者 ，你说是不？

在本章开始之前，有必要了解下MyBatis在SpringBoot下大致的工作过程是怎样的。

一、MyBatis相关类解释

******Mapper或者****Dao**

- 这是一个接口类，描述了某个数据访问的相关接口，例如： userMapper 是一个 tb_user 表的数据访问接口。
- com.wj.domain.mapper.UserMapper 是这个接口的完整类路径，这里叫做 mapperInterface 。
- 接口中定义了一些方法接口： int countByExample(UserExpample example) 、 int deleteByExample(UserExpample example) 等等，其中 countByExample 叫做方法名，它与 mapperInterface 一同组成了这个方法在某个连接配置中的唯一标示。

org.apache.ibatis.mapping.MappedStatement

- 它表示了Mybatis框架中， XML 文件对于sql语句节点的描述信息，包括 <select /> 、 <update /> 、 <insert /> 。
- 在初始化阶段，框架会将 XML 配置内容转为一个个 MappedStatement 对象实例。
- 在 XML 中， mapper.namespace.id 可以定位到唯一的一条 SQL 内容，这就是 MappedStatement 。所以，mapper.namespace 就是前面提到的 mapperInterface 。同样，在 XML 中通过 <include refid="_mapper.namespace.id" /> 可以使用其它 XML 中的 MappedStatement 内容。

org.apache.ibatis.binding.MapperProxy

它是 userMapper 的一个代理类，有三个主属性：

- SqlSession sqlSession： The primary Java interface for working with MyBatis. Through this interface you can execute commands, get mappers and manage transactions.
- Class<T> mapperInterface： 被代理接口的信息，比如 interface com.wj.springboot2demo.domain.dao.TbSearchManagerUserMapper
- Map<Method, MapperMethod> methodCache： MapperMethod 的缓存，是一个线程安全的 Map —— ConcurrentHashMap ，保存了每个mapper接口对应实现的sql命令和方法签名。

org.apache.ibatis.binding.MapperMethod

- MapperMethod 内部维护了两个final属性，都是 MapperMethod 内部类。

请输入关键字

栏目导航

- 关于我 (/about)
- 不止技术
  - 工程化应用（1 (/list?category
  - 技术学习/探索 (/list?category
  - 自娱自乐（2） categoryId=4)
- 还有生活
  - 随便写写（0） categoryId=3)
- 娱乐/放松（0） (/li categoryId=5)

点击排行

从零开发参数同步框架（二）
搜索引擎进阶——IK分词器扩
Nginx的nginx.conf配置部分
SpringBoot2从零开始（二）
Linux之相关工程化应用遇到
重温Java设计模式——建造者
Maven项目一键打包、上传、
从零开发参数同步框架（一）

标签云

java(6) (/list?tag=java)
参数同步(6) (/list?tag=同步)
netty(3) (/list?tag=nett
MyBatis(2) (/list?tag=MyBatis)
SpringBoot(2) (/list?tag=SpringBoot)
TCP/IP(2) (/list?tag=TC
maven(2) (/list?tag=ma

- SqlCommand command：它有两个字段， name 标识了 userMapper.countByExample 方法在 MappedStatement 配置内 id，type 标识的是 SQL 执行类型。
- MethodSignature method： 方法签名信息。具体如下：

```
/**
*
* 方法详细签名信息
*/
public static class MethodSignature {
    //返回值是否是VOID
    private final boolean returnsVoid;
    //是否返回多行结果
    private final boolean returnsMany;
    //返回值是否是MAP
    private final boolean returnsMap;
    //是否返回可枚举游标
    private final boolean returnsCursor;
    //返回值类型
    private final Class<?> returnType;
    //mapKey
    private final String mapKey;
    //resultHandler类型参数的位置
    private final Integer resultHandlerIndex;
    //rowBound类型参数的位置
    private final Integer rowBoundsIndex;
    ...
}
```

mongoDB(2) (/list?tag=mongoDB)
rpc(2) (/list?tag=rpc)
solr(2) (/list?tag=solr)
ssh(2) (/list?tag=ssh)
zookeeper(2) (/list?tag=zookeeper)
小程序(2) (/list?tag=小
搜索引擎(2) (/list?tag=
网络模型(2) (/list?tag=

站长推荐
微信小程序深入踩坑总结 (ar
Java网络编程之Netty学习（
工作中常用技术巩固——基础
搜索引擎进阶——solr自定义
Java网络编程之Netty学习（
Mac OS下安装Homebrew以
从零开发参数同步框架（五）
从零开发参数同步框架（一）

org.apache.ibatis.session.SqlSession

> The primary Java interface for working with MyBatis. Through this interface you can execute commands, get mappers and manage transactions.
>
> MyBatis最核心的一个接口类，通过它，可以实现SQL执行、事务管理等工作。

不过，在 MapperProxy 中注入的 SqlSession 对象是 org.mybatis.spring.SqlSessionTemplate ， SqlSessionTemplate 并没有自身对于接口 org.apache.ibatis.session.SqlSession 的逻辑实现，只是在内部代理了一个 SqlSession 的真正实现类—— org.apache.ibatis.session.defaults.DefaultSqlSession ， 真正的 Command 执行逻辑都在这个类里面实现。

SqlSessionTemplate

> 1. 是Spring管理的，且线程安全的。
> 2. Spring 事务管理模块维护了SqlSession在一整个连续的操作中的生命周期，包括SqlSession的创建、关闭、事务的提交、回滚等。
> 3. 它通过工厂方法 SqlSessionFactory 来创建自己所代理的 SqlSession 对象。
> 4. 由于它是线程安全的，所以它的一个实例对象可以被所有相关DAO或者Mapper共用。

二、初始化一个 MapperMethod 对象实例

在 Service 层使用的数据库查询接口Dao或者Mapper，是一个 Interface ， 并没有真正的实现类。例如我们使用的一个用户数量统计的 Mapper 类 tbSearchManagerUserMapper ， Spring 在具有依赖关系的地方给它注入的是一个代理类的实现 org.apache.ibatis.binding.MapperProxy 。

当执行 tbSearchManagerUserMapper.countByExample(example) 时，代码会进入代理对象的invoke方法 MapperProxy.invoke(Object proxy, Method method, Object[] args) throws Throwable 。

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        try {

            //method.getDeclaringClass()通过Method对象获得所属类的class信息
            if (Object.class.equals(method.getDeclaringClass())) {
                return method.invoke(this, args);

            //判断是否是默认方法
            //需要了解的内容：Method.isDefault，Method.getModifiers，以及Modifi
er
            } else if (isDefaultMethod(method)) {
                return invokeDefaultMethod(proxy, method, args);
            }
        } catch (Throwable t) {
            throw ExceptionUtil.unwrapThrowable(t);
        }
        //创建一个method对应的MapperMethod对象
        final MapperMethod mapperMethod = cachedMapperMethod(method);
        //执行查询
        //sqlSession 是一个`org.apache.ibatis.session.SqlSession`的实现类对象
        return mapperMethod.execute(sqlSession, args);
    }

    //根据`Method`创建一个`MapperMethod`实例，有一层缓存
    private MapperMethod cachedMapperMethod(Method method) {
        MapperMethod mapperMethod = methodCache.get(method);
        if (mapperMethod == null) {

            //根据被代理接口的Class信息、对应方法以及连接配置，创建一个MapperMethod实
例对象
            mapperMethod = new MapperMethod(mapperInterface, method, sqlSes
sion.getConfiguration());

            //保存到缓存中
            methodCache.put(method, mapperMethod);
        }
        return mapperMethod;
    }
```

```
1  public static class SqlCommand {
2
3      //xml标签的id，com.wj.springboot2demo.domain.dao.TbSearchManagerUserMappe
   r.countByExample
4      private final String name;
5      //insert update delete select的具体类型；在执行execute时会有用
6      private final SqlCommandType type;
7
8      //根据数据库配置信息、Mapper的接口类以及对应方法信息，构造一个SqlCommand对象
9      public SqlCommand(Configuration configuration, Class<?> mapperInterfac
   e, Method method) {
10         final String methodName = method.getName();
11         final Class<?> declaringClass = method.getDeclaringClass();
12         //根据`mapperInterface`和`methodName`共同组成`MappedStatement`的识别I
   D，获取一个`MappedStatement`对象
13         MappedStatement ms = resolveMappedStatement(mapperInterface, method
   Name, declaringClass,
14           configuration);
15         if (ms == null) {
16             if (method.getAnnotation(Flush.class) != null) {
17                 name = null;
18                 type = SqlCommandType.FLUSH;
19             } else {
20                 throw new BindingException("Invalid bound statement (not fo
   und): "
21               + mapperInterface.getName() + "." + methodName);
22             }
23         } else {
24             name = ms.getId();
25             type = ms.getSqlCommandType();
26             if (type == SqlCommandType.UNKNOWN) {
27             throw new BindingException("Unknown execution method for: " + n
   ame);
28             }
29         }
30     }
31
32     private MappedStatement resolveMappedStatement(Class<?> mapperInterfac
   e, String methodName,
33         Class<?> declaringClass, Configuration configuration) {
34         //生成MappedStatement标识ID
35         String statementId = mapperInterface.getName() + "." + methodName;
36
37         //如果之前已经加载， 则返回
38         if (configuration.hasStatement(statementId)) {
39             return configuration.getMappedStatement(statementId);
40         } else if (mapperInterface.equals(declaringClass)) {
41             return null;
42         }
43
44         //根据mapperInterface实现的接口，递归调用，获取MappedStatement
45         for (Class<?> superInterface : mapperInterface.getInterfaces()) {
46             if (declaringClass.isAssignableFrom(superInterface)) {
47                 MappedStatement ms = resolveMappedStatement(superInterface,
   methodName,
48               declaringClass, configuration);
49                 if (ms != null) {
50                     return ms;
51                 }
52             }
53         }
54         return null;
55     }
56     ...
```

三、执行查询动作

前面看到，在 `MapperProxy.invoke` 方法的最后一行就是执行查询动作 `mapperMethod.execute(sqlSession, args);`，并返回结果。

```
public Object execute(SqlSession sqlSession, Object[] args) {
    Object result;

    //MapperMethod中维护的一个SqlCommand对象，其中的type属性描述的就是本次sql的动作类型`SqlCommandType`
    switch (command.getType()) {
        case INSERT: {//插入
            Object param = method.convertArgsToSqlCommandParam(args);
            result = rowCountResult(sqlSession.insert(command.getName(), param));
            break;
        }
        case UPDATE: {//更新
            Object param = method.convertArgsToSqlCommandParam(args);
            result = rowCountResult(sqlSession.update(command.getName(), param));
            break;
        }
        case DELETE: {//删除
            Object param = method.convertArgsToSqlCommandParam(args);
            result = rowCountResult(sqlSession.delete(command.getName(), param));
            break;
        }
        case SELECT://查询
            //如果是查询，需要根据`MethodSignature`类型的属性`method`维护的内容进行进一步选择执行方法
            if (method.returnsVoid() && method.hasResultHandler()) {
                executeWithResultHandler(sqlSession, args);
                result = null;
            } else if (method.returnsMany()) {
                result = executeForMany(sqlSession, args);
            } else if (method.returnsMap()) {
                result = executeForMap(sqlSession, args);
            } else if (method.returnsCursor()) {
                result = executeForCursor(sqlSession, args);
            } else {
                Object param = method.convertArgsToSqlCommandParam(args);
                result = sqlSession.selectOne(command.getName(), param);
            }
            break;
        case FLUSH:
            result = sqlSession.flushStatements();
            break;
        default:
            throw new BindingException("Unknown execution method for: " + command.getName());
    }
    if (result == null && method.getReturnType().isPrimitive() && !method.returnsVoid()) {
        throw new BindingException("Mapper method '" + command.getName() +
" attempted to return null from a method with a primitive return type ("
                + method.getReturnType() + ").");
    }
    return result;
}
```

四、扩展 SpringBoot + Mybatis 多数据源

Spring Boot 在单数据源的情况下，会自动扫描 application 配置文件，而且自动将 DataSource 与 mapper.xml 目录进行关联，以及一些比如连接池 druid 的配置。

但是如果多数据源的配置， SpringBoot 默认无法将不同的 DataSource 与不同的 mapper.xml 目录进行关联。

> 事实上，SpringBoot 与 SpringCloud一样，都倡积极导微服务的概念和实践。微服务可以使不同的团队专注于更小范围的工作职责、使用独立的技术、更安全更频繁地部署。通常情况下，服务边界定义合理的微服务只会访问单一数据源。我猜测这就是 SpringBoot 在默认情况下，经过简单配置就可以实现数据源整合的原因，而多数据源的配置就需要用户自己去扩展实现。

前面已经说过，MyBatis对应Mapper进行查询的时候，实际数据库连接对象是 SqlSessionTemplate ，它由 SqlSessionFactory 对象创建而来，实际上 SqlSessionTemplate 内部维护了一个 SqlSessionFactory 实例。那么，我们可以自己写一个类，将创建 DataSource 、创建 SqlSessionFactory 和创建 SqlSessionTemplate 三步动作合并，同时将创建的 SqlSessionTemplate 与 mapper.xml 目录进行关联。

1、创建多数据源公用配置类

上一章《SpringBoot 2 从零开始（一）——项目启动 》(https://www.oomabc.com/articledetail?atclid=abaf81a2f5c246be8e643c45d7867888)提到过，在启动项目类上加了一个注解 @MapperScan(basePackages = "com.wj.springboot2demo.domain") 。

```
@MapperScan(basePackages = "com.wj.springboot2demo.domain")
public class Springboot2demoApplication {

    public static void main(String[] args) {
        SpringApplication.run(Springboot2demoApplication.class, args);
    }
}
```

其实，这个注解就是可以将 SqlSessionTemplate 与 mapper.xml 目录进行关联。所以，我们自定义配置类，只要能够根据配置文件创建 SqlSessionTemplate 就行了。

首先，我们定义一个公共配置类，这是一个好习惯，即使这类中没有任何内容。当然，我这个类是最终完成版本，所以里面并不是空的。

```
public class AbstractDataSourceConfiger {

    //与某个SqlSessionTemplate关联的XML目录
    //在配置文件中的位置和DataSource参数位置属于同一级别，参数名是：mapper-locations
    private String mapperLocations;

    //MyBatis的xml文件中使用的类的别名，位置与mapper-locations统一级别，参数名：type-aliases-package
    private String typeAliasesPackage;

    //根据dataSource创建一个SqlSessionFactory对象
    protected SqlSessionFactory createSessionFactory( DataSource dataSource) {
        SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
        bean.setDataSource(dataSource);
        // 添加XML目录
        ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
        try {
            bean.setMapperLocations(resolver.getResources(getMapperLocations()));
            bean.setTypeAliasesPackage(getTypeAliasesPackage());
            return bean.getObject();
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }

    public String getTypeAliasesPackage() {
        return typeAliasesPackage;
    }

    public void setTypeAliasesPackage(String typeAliasesPackage) {
        this.typeAliasesPackage = typeAliasesPackage;
    }

    public String getMapperLocations() {
        return mapperLocations;
    }

    public void setMapperLocations(String mapperLocations) {
        this.mapperLocations = mapperLocations;
    }
}
```

2、创建多数据源配置

代码如下：

```
//数据源A
//打开Bean注解方法的动态代理功能：该类中带有注解`@Bean`的方法，都会被动态代理，调用该方法会返回同一个实例；本质上还是`@Component`
@Configuration
//这里就是将`SqlSessionTemplate`与`mapper.xml`目录进行关联；
//其中，`sqlSessionTemplateRef`配置的就是下面带有`@Bean`的方法，该方法最终返回了`SqlSessionTemplate`实例，而且是单例的
@MapperScan(basePackages = { "com.wj.springboot2demo.domain.searchmanager" }, sqlSessionTemplateRef = "searchManagerSqlSessionTemplate")
//这个注解的作用是，将`application`配置文件的属性绑定到了当前类，这里的作用是绑定公共类`AbstractDataSourceConfiger`中的两个属性。
@ConfigurationProperties(prefix = "spring.datasource.druid")
public class SearchManagerDsConfiger extends AbstractDataSourceConfiger {

    //当前方法返回的是一个`DataSource`实例，而且是`Spring`动态代理的，其他地方通过注解@Qualifier注入
    @Bean(name = "searchManagerDataSource")
    @Primary // 有一个默认的DataSource加此注解，而且只能有一个
    // prefix值必须是application.properteis中对应属性的前缀
    @ConfigurationProperties(prefix = "spring.datasource.druid")
    public DataSource userDataSource() {
         DataSource dataSource = DruidDataSourceBuilder.create().build();
         return dataSource;
    }

    @Bean
    public SqlSessionFactory searchManagerSqlSessionFactory(@Qualifier("searchManagerDataSource") DataSource dataSource) throws Exception {
        //调用公用方法，根据 DataSource 创建 SqlSessionFactory 对象
        return createSessionFactory(dataSource);
    }

    //根据 SqlSessionFactory 创建一个 SqlSessionTemplate 对象实例，这里的方法名就是 类注解 @MapperScan 中 sqlSessionTemplateRef 的应用
    @Bean
    public SqlSessionTemplate searchManagerSqlSessionTemplate(
            @Qualifier("searchManagerSqlSessionFactory") SqlSessionFactory sqlSessionFactory) throws Exception {
        SqlSessionTemplate template = new SqlSessionTemplate(sqlSessionFactory); // 使用上面配置的Factory
        return template;
    }

}

//数据源B
@Configuration
@MapperScan(basePackages = { "com.wj.springboot2demo.domain.ho" }, sqlSessionTemplateRef = "hoSqlSessionTemplate")
@ConfigurationProperties(prefix = "spring.second-datasource.druid")
public class HoDsConfiger extends AbstractDataSourceConfiger {

    @Bean(name = "hoDataSource")
    @ConfigurationProperties(prefix = "spring.second-datasource.druid")
    public DataSource hoDataSource() {
        DataSource dataSource = DruidDataSourceBuilder.create().build();
        return dataSource;
    }

    @Bean
    public SqlSessionFactory hoSqlSessionFactory(@Qualifier("hoDataSource") DataSource dataSource) throws Exception {
        return createSessionFactory(dataSource);
    }
```

```
54
55      @Bean
56      public SqlSessionTemplate hoSqlSessionTemplate(
57              @Qualifier("hoSqlSessionFactory") SqlSessionFactory sqlSessionF
    actory) throws Exception {
58          SqlSessionTemplate template = new SqlSessionTemplate(sqlSessionFact
    ory); // 使用上面配置的Factory
59          return template;
60      }
61
62  }
```

最终的 `application.yml` 配置文件如下：

```
server :
    port : 8081

spring :
    datasource :
        druid :
            filters : stat
            driver-class-name: com.mysql.jdbc.Driver
            #基本属性
            url: jdbc:mysql://192.168.50.42:3306/search_manager?useUnicode=true&
characterEncoding=UTF-8&allowMultiQueries=true
            username: test
            password: test
            #配置初始化大小/最小/最大
            initial-size: 1
            min-idle: 1
            max-active: 20
            #获取连接等待超时时间
            max-wait: 60000
            #间隔多久进行一次检测，检测需要关闭的空闲连接
            time-between-eviction-runs-millis: 60000
            #一个连接在池中最小生存的时间
            min-evictable-idle-time-millis: 300000
            validation-query: SELECT 'x'
            test-while-idle: true
            test-on-borrow: false
            test-on-return: false
            #打开PSCache，并指定每个连接上PSCache的大小。oracle设为true，mysql设为fals
e。分库分表较多推荐设置为false
            pool-prepared-statements: false
            max-pool-prepared-statement-per-connection-size: 20
            type-aliases-package : com.wj.springboot2demo.domain.model
            mapper-locations : classpath*:mapper/searchmanager/*.xml

    second-datasource :
        druid :
            filters : stat
            driver-class-name: com.mysql.jdbc.Driver
            #基本属性
            url: jdbc:mysql://192.168.50.42:3306/ho?useUnicode=true&characterEnc
oding=UTF-8&allowMultiQueries=true
            username: test
            password: test
            #配置初始化大小/最小/最大
            initial-size: 1
            min-idle: 1
            max-active: 20
            #获取连接等待超时时间
            max-wait: 60000
            #间隔多久进行一次检测，检测需要关闭的空闲连接
            time-between-eviction-runs-millis: 60000
            #一个连接在池中最小生存的时间
            min-evictable-idle-time-millis: 300000
            validation-query: SELECT 'x'
            test-while-idle: true
            test-on-borrow: false
            test-on-return: false
            #打开PSCache，并指定每个连接上PSCache的大小。oracle设为true，mysql设为fals
e。分库分表较多推荐设置为false
            pool-prepared-statements: false
            max-pool-prepared-statement-per-connection-size: 20
            mapper-locations : classpath*:mapper/ho/*.xml
```

至此，SpringBoot2 + MyBatis 多数据源配置就完成了。

相关文章

SpringBoot 2 从零开始（一）——项目启动
SpringBoot2从零开始（三）—— rabbit MQ
Nginx的nginx.conf配置部分解释 (articledetail?
从零开发参数同步框架（六）—— 简版配置中心